National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

Environment Editor

by

Qinlin Tang

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Fall 1992

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77235-2

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Qinlin Tang
TITLE OF THESIS: Environment Editor

DEGREE: Masters of Science
YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis not extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) . . . . . .

Permanent Address:
#42, 11008-88Ave.
Edmonton, Alberta
Canada, T6G 0Z2

Date: April 30, 92

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Environment Editor** submitted by **Qinlin Tang** in partial fulfillment of the requirements for the degree of Masters of Science.

Dr. Mark Green(Supervisor)

Dr. Jim Hoover

Dr. Allen Carlson

Date: April 30, 92

Abstract

Researchers believe virtual reality environments with aesthetic implications will open a new dimension for arts by changing the way art is created and experienced. This thesis is concerned with the issues related to virtual reality as a new artistic medium.

RAG, which is a recently proposed artistic virtual reality system, has been introduced to explore methods and tools for building virtual reality applications in arts. The environment editor, which is one of the components of this system, has been implemented in this thesis.

The environment editor is a software tool that provides artists with an easy-to-use design tool where the artists design and preview their art works: the virtual environments. The objects in a virtual environment are organized in a hierarchical manner. The DataGlove is used in the environment editor as a 3D interaction device for directly manipulating the objects in a virtual environment.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Mark Green, for his support and inspiration. His vision and dedication lay the ground work of this thesis. I am also grateful to the members of my committee - Dr. Jim Hoover and Dr. Allen Carlson for their evaluation of this thesis. I would like to acknowledge Chris Shaw for the videotaping and Robert Lake for providing the nice working environment in the graphics lab. Exceptional thanks are due to Yigong Wang, whose advice guided this thesis from its onset. Without his support, my task would have been more difficult. Above all, I am deeply indebted to my parants, brother and sister for their love, understanding and patience, for staying so close to me, and for being such good listeners.

To my parents

Contents

1 Introduction 1

1.1 What Is *Virtual Reality*? 2

1.2 Survey of Previous Work on Virtual Reality 4

1.3 The Future Possibility of Virtual Reality 7

1.4 RAG - A Virtual Reality Application in Arts 8

1.5 Thesis Outline 10

2 A Virtual Reality Environment - RAG 11

2.1 Terminology 11

2.1.1 Event and Behavior 11

2.1.2 Object, Object Prototype and Object Instance 12

2.1.3 Space 12

2.2 System Architecture of RAG 13

2.3 Object Modeling Language (OML) 15

2.4 Object Compiler 15

2.5 Environment Editor 16

2.6 Environment Compiler 17

2.7 The Viewing Program 17

2.7.1 Event Determination 18

2.7.2 Behavior Execution 19

2.7.3 Image Generation 20
2.7.4 Sound Production 20
2.8 Delivery Vehicle 21
2.8.1 Delivery Vehicle Hardware 21
2.8.2 Delivery Vehicle Software 22

3 Environment Editor and Its Implementation **24**
3.1 The Working Space 24
3.1.1 Assumptions 25
3.1.2 Setup of the Working Space 25
3.2 Overview of Environment Editor Design 27
3.2.1 Functions of Environment Editor 27
3.2.2 User Interface Layout of the Environment Editor 29
3.3 Space Editor 31
3.3.1 Space Boundary Editing 31
3.3.2 Space Content Layout 32
3.3.3 Event-Behavior Table 33
3.3.3.1 Structure of Event-Behavior Table 34
3.3.3.2 User Interface to the Event-Behavior Table 34
3.3.4 Space Modification 36
3.4 Object Editor 37
3.4.1 Object Hierarchy in FDB 37
3.4.2 Generation of Object Instances 39
3.4.3 Master Editing and Instance Creation 41
3.5 Environment Editing Subsystem 42
3.5.1 Environment Creation and Modification 43
3.5.2 Other Operations 43

4 Manipulation of 3D Objects **44**

4.1 Methods for Editing Objects 44

4.1.1 DataGlove 45

4.1.1.1 Editing Path3 Parameters 45

4.1.1.2 Setting Transformations 46

4.1.2 Grids and Potentiometers 48

4.1.3 Buttons and Scrollable Lists 50

4.1.4 DataGlove vs. Potentiometers 51

4.2 Grabbing Operations 51

4.2.1 The Problems 52

4.2.2 The Grabbing Algorithm 53

4.3 Space Boundary Constraints 56

5 Conclusions and Future Work **60**

A The Database Schema **63**

B User Manual **66**

B.1 Environment Creation 66

B.1.1 Use of Object Editor 70

B.1.2 Use of Space Editor 71

B.1.3 Environment Layout 72

B.2 Environment Modification 72

List of Figures

1 The system architecture of RAG. 14

2 The delivery software. 22

3 The setup of the environment coordinate system. 26

4 The layout of the environment editor. 30

5 The event-behavior table. 35

6 The high level structure of the environment database. 38

7 The low level structure of the environment database. 39

8 The grabbing algorithm. 54

9 The high level menu structure of the environment editor. 67

10 The menu structure of the object editor. 68

11 The menu structure of the space editor. 69

Chapter 1

Introduction

Over the past few years computer hardware costs have fallen dramatically; software costs have also decreased, although less rapidly than hardware. What remains to be addressed is the optimization of the interface between the user and the computer and the maximization of user efficiency. The concept of virtual reality provides the user with a more communicative interface with the computer. Virtual reality environments being developed are aimed at making computers more responsive to human modes of communication including gestures, postures, speech and sound.

Researchers believe that virtual reality environments with aesthetic implications will open a new dimension for art by changing the way art is created and experienced [Kru83]. Artistic virtual reality environments can perceive human activities and respond through aesthetically pleasing images and sounds. The beauty of an art work in the virtual reality form is experienced through the interaction process between the viewer and the virtual reality. The blueprints for virtual reality in art have been drawn up. But what is the right approach to building virtual reality applications in the arts? This thesis is a step towards providing a tool for constructing artistic virtual reality environments, the model of which is conceived at University of Alberta, and explores the issues related to this process.

1.1 What Is *Virtual Reality*?

Many ideas of virtual reality first came from science fiction. In science fiction, virtual realities are realistic models of the real world created by computers and composed of sensing, display and control systems. In principle, the virtual reality environment in science fiction can respond to a viewer's position, voice, smell and taste. It can also respond to the viewer's movement, posture and gesture.

However, technology won't catch up with the science fiction writers' imagination for several years. With the recent hardware advances, such as the advent of the DataGlove and EyePhone, computer scientists are in the position to fully explore virtual reality applications. The DataGlove and EyePhone are considered to be the standard input and output devices in virtual reality. The DataGlove is an instrument using fiber-optics to measure bend degree of a single finger joint and a sensor to determine the hand's position. The DataGlove has exciting potential in virtual reality environments as it combines the precision, control and agility of the human hand. The EyePhone is a piece of equipment that is mounted in front of the eyes. It uses two television sets, one for each eye. Each eye is provided with a slightly offset view of the same image generated by computer. By adding depth perception, the user can see stereo pictures through the EyePhone. There is also a sensor on the top of the EyePhone that is used to trace the user's head position. With the aid of this sensor, the user can enjoy the illusion of scanning a virtual panorama as he moves his head. Thus, the EyePhone achieves a dynamic stereoscopic effect of a 3D environment. The combination of the DataGlove and EyePhone can provide the user with the sense of reality in the virtual environment. This kind of environment is called *Virtual Reality.*

From the computer scientist's point of view, virtual reality is a new type of user interface. The evolution of the user interface technology has occurred in three stages. Before the 1980's, user interfaces were basically command line interfaces. The input devices such as keyboards and single window display screens were usually restricted

to one dimensional text input. Through out the 1980's, user interface design has been dominated by the two dimensional metaphor. This metaphor consists of the multi-window display screen and two-dimensional input device - the mouse. The user interface design technology at this stage is suitable for office tasks like documentation processing and programming [Gre 7].

However, a wide range of applications, particularly those that can be represented in 3D, still call for a more advanced user interface technology. For example, scientists want to visualize a huge amount of data from satellites; architects want to walk through the prototypes of their designs to get feedback before actually constructing them; chemists want to "be there" inside a molecule to do their experiments. Obviously, 3D user interface technology has to be explored to satisfy the needs of these types of applications. Virtual reality was born for this purpose, providing a means for the user to interact with the computer in a more intuitive and direct format than ever before. Therefore, virtual reality is also a 3D user interface that places the user in the 3D environment and allow the user to directly manipulate the environment.

Virtual reality has three key components: image viewing, behavior and interaction. Visual images help the user to interpret the information being presented by the computer. These images may represent real objects, such as building frames, or abstractions such as patterns of fluid flow, or micro-world phenomena, such as protein structure. Virtual reality is created, it does not just mimic reality. It can go beyond reality, by modeling in concrete form abstract entities such as mathematical equations. Real-time image production is essential to virtual reality since the virtual environment needs to provide the user with the visual sense of response to his or her interaction. These images behave the way the objects or abstractions they represent would behave. Behavior modeling needs heavy computation in many cases, because it often involves solving extensive sets of equations over and over again. Finally, the user interacts with a virtual reality environment in a direct way by moving around

and observing the environment from many different angles, pointing and grabbing virtual objects in the environment, even by talking with the environment.

1.2 Survey of Previous Work on Virtual Reality

Mimicing reality such as in flight simulation systems foreshadowed virtual reality. Computers simulate the sound, force and motion that approximate the aerodynamic behavior of an airplane, and computers also provide the images corresponding to the movement of the airplane. The pilot, therefore, can be trained in such a simulated airplane flying over virtual terrain. The flight simulation system shows the ability of modern computer technology to mimic reality. It is natural to speculate if it is possible to extend the simulation capabilities of computer technology to solve other problems.

The Walkthrough project conducted by Frederik P. Brooks and his colleagues at University of North Carolina at Chapel Hill is one of the early virtual reality systems [Bro86]. The Walkthrough project aims at providing a tool in which virtual buildings, designed but not yet constructed, can be explored by "walking through" them in order to refine the specification. The system allows the viewer to navigate through the virtual building, experiencing new views instead of viewing the building from a static position, or from a pre-planned cruise path in order to obtain a deeper understanding of the spatial relations of the architecture.

The system structure of the Walkthrough system consists of six modules. The builder subsystem is responsible for constructing and modifying the building model. The master model is a relational database system containing the building's components which are inputed by the Builder module. The master model is organized in an easy-to-manipulate form. The working model is another database constructed for fast view generation. The viewer instancer module makes views from the working module

and also drives the display devices. The view specifier module is the user interface of the system. Finally, the toolbox module contains algorithms and data manipulation routines needed by the other modules.

Traditional CAD-CAE systems for building architecture basically assist the designer in producing the building structure. The significance of this work is that it helps the designer to visualize and explore buildings in a 3D environment while he is in the design stage. However, the system does not provide enough interactive capabilities for the viewer to directly manipulate the prototype of the buildings during the viewing process.

Virtual reality is not restricted to simulating a model of the real world, but can also be a touchable model of the systems that cannot ordinarily be touched such as the micro-world. Why do we need to see and feel a untouchable world? Psychological research shows that adding a haptic element to a displayed system can enhance the perception of the described system [FPJB+90]. For instance, imagine a chemist examining two molecules: an enzyme and its substrate. He knows the structure of both the enzyme and its substrate and the two molecules are displayed on the screen. The chemist wants to find out what part of the enzyme interacts with what part of the substrate. Imagine being able to feel the shape of the enzyme, the chemist probes the active point of the enzyme with his finger. The enzyme thus exerts a strong chemical attraction on the substrate. The chemist can instantly feel the pull of the interatomic forces that join the two molecules. Can this imagination be true?

The GROPE system, which is aimed at achieving the effects described above, is a haptic display system for molecule chemistry that has been developed at UNC. The GROPE system has three generations. The first generation GROPE-I is a 2D force feedback system for teaching about force fields [BB71] in Physics. The second generation: GROPE-II is a 3D system which can evaluate the molecular forces [Bro77]. The new generation, GROPE-III enables chemists to see and feel the interactions between

molecules, such as an enzyme and its substrate [FPJB+90].

The Walkthrough system and the GROPE system basically use gestures to interact with the environment. In what other ways might a user want to interact with a virtual reality besides non-verbal manipulations of the virtual world such as gesturing? Talking to a computer seems to be more convenient. More than ten years ago Richard A. Bolt of the MIT demonstrated the feasibility of voice recognition in computer interaction [Bol80] [Bol81].

Bolt's voice recognition experiment was conducted in a media room that had a wall-sized screen for display and a number of input devices for gesture input. The experiment was designed to use speech to manipulate an information database instead of typing in the commands from a keyboards. As the computer needs more precise description for the pronouns such as "this" or "that", some pointing gestures were designed to aid the computer in handling the pronouns. Bolt's system is an experiment concerning direct means of computer interaction which is one aspect of the virtual reality.

Voice recognition systems have become more and more sophisticated through the 1980's. IBM Corporation's Thomas J. Watson Research center has built a system with a vocabulary of 20,000 words. This system is able to distinguish between words with very similar pronunciation. However, the technology for voice recognition has not yet been integrated into most virtual reality environments.

The ability of modern computers to create aesthetically pleasing images and to orchestrate the sound, animation and interaction between the user and the computer shows the great potential of the applications of virtual reality in arts. The earliest application of virtual reality in arts is that of Myron Krueger. In his book "Artificial Reality" [Kru83], he proposed that virtual reality, which he called artificial reality, had the potential to become a new aesthetic medium by changing the viewer's role from passive observer to active participant.

An experimental system called VIDEOPLACE was conceived in 1969 by Krueger's group, and has been undergoing continuous implementation since 1974 [MWKH85] [Kru91]. The VIDEOPLACE system is built in a dark room with a large video projection screen on one of the walls and a video camera mounted below the screen. The user stands in front of the video projection screen during the experiments. The user's image is viewed by the video camera and joined with other graphic objects displayed on the projection screen. The graphic objects have behaviors which react to the movements of the user's image in real-time.

The latest version of the VIDEOPLACE system is able to create the real-time interactions between two people in different locations. A second system called VIDEODESK has been developed. It consisted of a light table with a camera mounted above and aimed down at the table's surface where the computer monitor resides. An image of the user's hands is displayed on the screen. VIDEOPLACE and VIDEODESK are connected together so that the interactions can be shared by the users in the two environments [Kru91].

The VIDEOPLACE system is quite different from other virtual reality systems in the way that the user does not have to wear any interaction devices in order to interact with the virtual reality as other systems do. This demonstrates an alternative approach to the design of virtual reality applications.

1.3 The Future Possibility of Virtual Reality

Looking back in history, when television first came out, it was viewed as a toy that artists did not pay much attention to. In the past decades, however, television has exerted a great influence on our life. Virtual reality is only in its early stage. People have not figured out how wide the applications of this technology can be. But like television, the new technology will predictably exert a tremendous influence on our

daily life in the near future.

In an ideal virtual reality environment, the user will have various ways to communicate with this environment. He or she can see the environment, can hear sound from the environment, or even can smell or taste virtual objects in the environment. Virtual reality will be a new universe where some aspects of reality are enhanced since many of the constraints and limitations of reality can be overcome while others are simply deleted.

Some people imagine that one day humans will be able to create a perfect world to live in by using virtual reality technology. They might be able to stay at home and simply press a button to switch on virtual reality. Then they might travel to a tropical beach, have a meeting with colleagues there or enjoy the sun. People do not actually leave their real homes when all these things happen. They might even be able to walk into the television and interact with the characters on a television show. They might also be able to visit a virtual moon while experiencing the feeling of wandering around the real moon. It is also imagined that one day people can go back in time and experience everything in the past. The experience that the user gets from virtual reality is like being able to take a trip without actually having to go there in some sense.

The concept of virtual reality has opened up a rich variety of computer applications. Virtual reality can be used in science, arts, the study of history and many other fields. Both computer technology and human imaginations will push this new technology in directions that are beyond foresight.

1.4 RAG - A Virtual Reality Application in Arts

Virtual reality opens up a new dimension for arts as this technology can be used to construct visually and sonically pleasing 3D environments with which viewers can

directly interact. The responsive environment develops a new relationship with its viewer and its art. In the traditional art forms, a viewer takes a passive role and solely enjoys the visual and aural beauty of an artist's work. However, a viewer is no longer passive in virtual reality. He or she is able to manipulate objects in the artistic environment and experience different versions of beauty based on a rich variety of alternatives composed and provided by the artist. The interaction will take the viewers into an exploration of their own senses and mental processes. The viewers' actions complete the piece of art work. In some sense, the viewers also participate in the creation process of the virtual realities. As a result, virtual reality is a new aesthetic medium for artists and viewers.

The responsive environment described above is named Responsive Art Gallery, which has two features. One is that it keeps a link to the traditional art forms by providing aesthetically pleasing environments, and the other is that it adds an interactive and responsive element which allows viewers to participate in the creation of the art works. We use RAG to stand for Responsive Art Gallery. The system design of RAG has been proposed recently at University of Alberta [Gre90]. The aim of constructing this system is to explore practical methods for creating artistic scenarios for virtual reality, and to build the basic computer software and hardware that will support various future virtual reality projects.

Three groups of people - computer scientists, artists and users (or viewers) are destined to work as a team in completing RAG. The computer scientists provide a set of software tools that support the construction of RAG, and also provide the basic vocabulary for building scenarios of RAG.

The artists use the tools and the vocabularies to design and construct scenarios of RAG. The design and construction tasks include specifying the geometric and behavioral properties of the objects in the environment, and establishing the connection between events and behaviors.

When the artists finish their tasks, the users will be able to enter RAG and explore the beauty of art work in the gallery. The interaction between a user and the gallery starts when he or she triggers pre-arranged events and the environment responds to these events.

RAG is not just a single piece of art work, but a powerful medium that can be used to compose a wide variety of artistic scenarios. The relationship between the viewer's actions and the environment's response is composed by the artists. A scenario of RAG can be realistic or fantastic. The artists' imagination is the only constraint.

1.5 Thesis Outline

This thesis is concerned with the design and implementation of the environment editor component of RAG. This work is part of RAG. The interfaces between components are described in [Gre91].

The remainder of this thesis is organized in the following way. Chapter 2 elaborates on the system architecture of the art gallery by presenting the concepts involved and interfaces to the environment editor. Chapter 3 presents the design and implementation of the environment editor in detail. The work is focused on the editor's functionality and the user interface part. Chapter 4 describes several techniques used to manipulate 3D objects in a virtual reality environment. These techniques are intensively used by the environment editor. Chapter 5 discusses some future features of the environment editor and concludes this thesis. The appendices include the database schema for the environment editor and a user manual describing how to use the editor.

Chapter 2

A Virtual Reality Environment - RAG

The preceding chapter has given several definitions of *virtual reality* from different perspectives. This chapter will give an example of such a virtual reality environment that explores this new area further and lays the foundation of this thesis. This virtual reality environment is named *RAG* that is proposed in [Gre90]. Since the environment editor, which is the main concern of this thesis, is part of this environment, it is important to discuss the system architecture in detail.

2.1 Terminology

Before we proceed to discuss the system architecture of RAG, it is necessary to give the definitions of some terms used in the rest of this thesis.

2.1.1 Event and Behavior

An event is a signal that informs other parts of the environment that something has happened. A name is associated with each event to identify the event. Usually, the

viewer's action, or changes in the geometry or position of objects in the environment may cause events to occur.

Behavior is the task that an object performs in response to a certain event. It is usually associated with objects(Refer to the definition of object).

2.1.2 Object, Object Prototype and Object Instance

Entities that have geometrical and behavioral properties in the environment are called objects. An object must have the geometric property, but may or may not have the behavioral property.

An object's geometric property defines the object's shape in the environment. An object's geometry can be both static and dynamic during its life time in the environment. An object's appearance can change in response to certain events.

The behavioral part of an object definition describes how the object responds to the events that occur in the environment. Each behavior has a name, a set of parameters, and a procedure that performs the action for the behavior.

The geometry and behavior of an object originally defined by the computer scientists forms an object prototype. An object prototype is the abstract description or the skeleton of the object in a parameterized form. An object instance is a concrete entity derived from its prototype whose parameters have been customized by the artist.

2.1.3 Space

A space is defined as a region of the environment that can be treated as a unit. Space has the following properties. For each space, there is a constant mapping between events and object behaviors, but the mapping can vary from one space to another. A space by itself has no visible representation. It only becomes visible when an object is added into the space. Space is a closed region defined by a collection of polygons.

2.2 System Architecture of RAG

The system architecture of RAG is presented in Figure 1. As mentioned in Chapter 1, there are three groups of people involved in the virtual reality environment of RAG, therefore, RAG is built in three layers. The computer scientist works at the lowest layer. Their main concern is to construct the basic building blocks of RAG. The descriptions of objects and spaces are written in a specification language called *Object Modeling Language* or OML. The OML program will be processed by the OML compiler to produce a set of vocabularies that define prototypes of objects and spaces. These prototypes are the abstract description of objects and spaces that will be instantiated into concrete objects and spaces.

The artists work at the middle layer to design the environment with a software tool called the *environment editor*. The prototypes of objects and spaces are the materials for constructing the environment. Their geometric and behavioral parameters have to be edited through this environment editor. The object prototypes can be constructed into composite objects by using a master-instance relationship that is a standard graphics technique. This task is also done by using the editor. In addition, the editor provides a user interface for artists to easily associate events with object behaviors. Later on, an object's behavior is automatically triggered when its associated event occurs. After the artist finishes his/her design, the virtual art gallery environment is stored in the *frame based database* system (FDB) [Dep90a].

The data structure of FDB used to represent the environment in the editor is tuned to facilitate editing and manipulating objects or spaces, while the data structure used in the viewing program, to be described below, must support fast display and efficient interaction with the environment. Therefore, another tool, called the *environment compiler*, is used to convert the environment stored in FDB into the data structure for the viewing program, which is broken into two subsets of data structures: one for displaying the environment, and the other for event determination

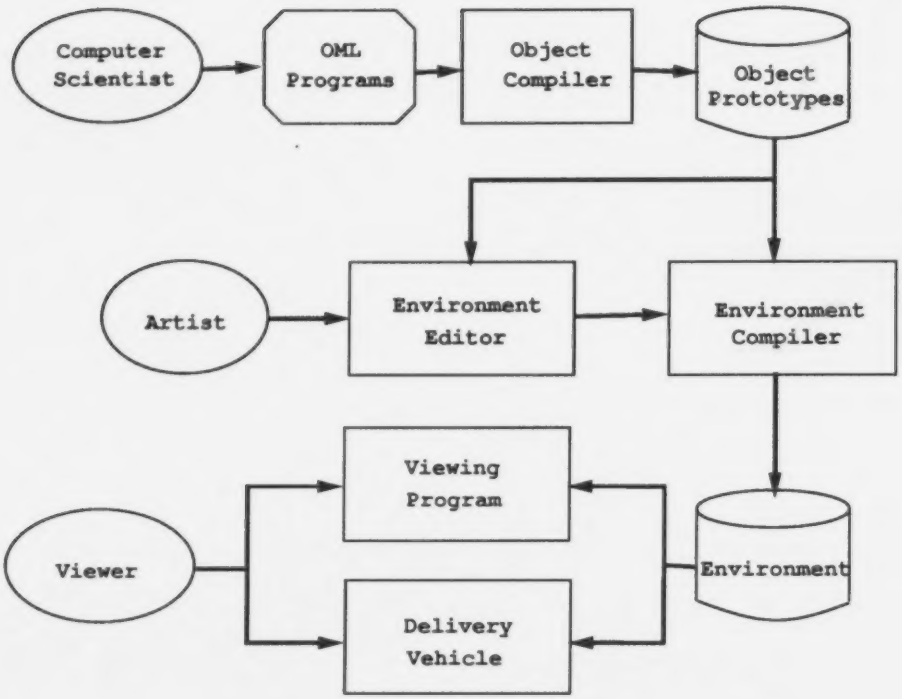

Figure 1: The system architecture of RAG.

and behavior execution.

The top layer of the system which includes the viewing program and delivery vehicle is mainly operated by the user or the viewer of the art gallery. The viewing program is responsible for generating images of the visible parts of the environment to the viewer. The viewing program also needs to determine the events that have been generated by the user or the objects in the environment and then execute the associated object behaviors to response to these events. As a sequence of event determinations and behavior executions gradually and gracefully change the images, the viewers experience a variety of beautiful art works.

A delivery vehicle is used to place the user in the art gallery environment so that the user can virtually view or even interact with the environment. The virtues of employing a delivery vehicle are that more than one user could view and interact with this art gallery at the same time, and that these users can control their viewing

positions and moving speeds individually.

RAG proposed by University of Alberta aims at exploring the methods and tools for building and editing highly interactive virtual reality environments, while many existing virtual reality environments are more or less a viewing system of static polygon databases [CB+90].

2.3 Object Modeling Language (OML)

The object modeling language is a high-level textual language which is built on top of the C language. OML is used to define the geometric and behavioral information of the object and space prototypes used in RAG.

The structure of a program written in OML is basically a series of object or space prototype definitions. For the definition of each object, there are three sections: parameter section, geometry section and behavior section.

The parameter section contains the names and types of all the parameters used in the geometry and behavior sections. The geometry section consists of a sequence of statements of the OML geometry modeling language. These statements are used to construct the object's geometry. The behavior section is specified using a behavior description language based on the relation model [Sun92]. The OML language has been conceived and is being implemented at University of Alberta [Gre92].

2.4 Object Compiler

The object descriptions written in OML are converted into object prototypes by the object compiler. An object prototype provides all the necessary information to generate and modify an instance of the object. The information in a prototype is mainly retrieved by the environment editor and environment compiler in the form of procedure calls. An object prototype only defines the "skeleton" of the object. An

instance of the object is created from the skeleton by the environment editor.

Based on the three sections of an object described in OML, the object compiler generates three groups of procedures that will be used by the environment editor and the environment compiler. The first group of procedures, called the parameter editor, provides functions to assign values to object parameters defined in OML's parameter section. The environment editor provides a simple interface to edit these parameters. Therefore, the procedures in this group are frequently called by the environment editor.

The second group of procedures are used to construct the object's geometry in an environment. The objects are produced when their construction procedures are called with the parameters set up by the first group.

The third group of procedures is responsible for object behavior. The environment editor provides an easy-to-use user interface to associate an event with an object behavior. The system will automatically call the procedure defining the behavior when the event corresponding to the behavior occurs.

2.5 Environment Editor

The environment editor is a software tool that provides artists with a user-friendly interface to assist them to create or modify the virtual art gallery environment. The input to the environment editor is a file containing object prototypes. The output of the editor is the environment database which contains all the information of the objects and spaces in the environment. The environment database is stored using the FDB.

This thesis describes the design and implementation of the environment editor in great detail starting in the next chapter. Therefore, a detailed description of it is not presented in this chapter.

2.6 Environment Compiler

The data structures produced by the environment editor are organized into an easy-to-edit form. The efficiency of displaying and interacting with the environment has not been taken into account in these data structures. It is critical to have data structures supporting the fast display of the environment since the user has to interact with the environment in real-time. Delays in displaying objects will cause the user to be physically uncomfortable in the virtual reality environment. The environment compiler's main task is to convert the data structures produced by the environment editor into the data structures suitable for the viewing program.

The environment compiler generates two sets of data structures. One set is used in displaying the environment, and the other set is used for interacting with the environment including event determination and behavior execution.

The data structures for display are based on the BSP tree technique that partitions the environment into a number of well-behaved regions where the hidden surface problem can be solved efficiently. The event determination also uses a BSP tree constructed from the spaces in the environment. With the BSP tree, it is easy to determine events, such as when a viewer leaves or enters a space, by simply traversing the tree to find the space that the viewer is currently occupying.

2.7 The Viewing Program

The viewing program has four major tasks that deal with event determination, behavior execution, image generation and sound production. The user directly interacts with the environment through the viewing program running on the delivery vehicle. In addition to techniques specific to virtual reality, the viewing program needs to use advanced 3D graphics techniques such as hidden surface removal, illumination models, and computer animation. This section only gives an overall idea of how this

viewing program works.

2.7.1 Event Determination

There are three sources of events: the viewer, the environment itself and the objects within the environment. The user's actions could generate some predefined events. The environment also has a few events that occur independently of the viewers and the objects in the environment. Objects also may generate events to trigger the behaviors of other objects.

The actions that a viewer can perform to generate events can vary. However, for simplicity, the system includes the following user's actions: the viewer's body posture and body movement within a space, leaving or entering a space, grabbing an object and various other pre-defined hand gestures. All these actions are treated as the source of events. When one of these events occurs, the environment makes an appropriate response to it. When the viewer moves into or out of one of the spaces, the environment may respond dramatically.

The environment is designed to be a four dimensional space in a certain sense. Time is one of the dimensions taken into account. A calendar and a clock are used in the environment to simulate the change of seasons and the passage of time. The environment calendar generates events periodically to inform the change of seasons. The clock's ticks yield another set of events to broadcast the time change from moment to moment. To these environment events, autonomous objects within the environment may respond by showing different behaviors. Thus the environment shows different phenomena as the environment evolves.

The events generated by an object can vary depending on the space that the object is in, the situation when the event happens and the relations among objects. For example, when a shark approaches a fish, an event will be generated and the fish will respond by moving away from the shark. This type of event is defined in

the process of defining object behaviors. An object behavior that may cause other objects' response corresponds to such an event.

The events generated in the environment are queued in an event list. The environment events are handled by maintaining a list of the objects that are supposed to respond to the events. Thus, when an environment event occurs, the object list is examined and the object behavior for the event is performed. Object events are determined by an event recognition procedure. When an object executes a behavior, this procedure is invoked to check whether an event has occurred. Then the generated event is added to the event list.

Determination of the events generated by the viewer are slightly more difficult than the other two types of events. The events generated by hand gestures are determined in the following way. Each space is associated with a set of hand gestures that can happen in the space. Each gesture can generate a list of events. The relationship between the gesture and its events are represented by a standard gesture table for each space. The DataGlove software determines if the user is making any of the recognized gestures, and then a gesture table lookup is performed to determine which events are occurring.

In order to handle arbitrary body motion, a set of event recognition procedures is attached to each space. Each procedure recognizes a particular type of event. As mentioned above, these events include space entry and space exit events. The BSP tree techniques can be used to determine a viewer's position, and thus space entry or space exit events can be determined. When an event is determined, it is added to the event list that is used for behavior execution described in the next section.

2.7.2 Behavior Execution

The event determination step produces a list of the events that have occurred. The environment editor sets up a mapping table between the events and the object be-

haviors that respond to these events. This table is called the event-behavior table. Behavior execution involves scanning through the list of events and for each event executing the corresponding behavior procedure defined in the event-behavior table.

2.7.3 Image Generation

This module is responsible for producing images of objects in the environment. The objects in the environment are represented by a group of BSP trees and an object tree generated by the environment compiler. There is only one object tree produced by the environment compiler for each environment. Each object in the environment has one or more BSP trees generated by the environment compiler. The image generation module provides procedures to handle the object tree and BSP trees. The image is produced by traversing the object tree based on the viewer's current position. During the traversal of the object tree, the objects invisible to the viewer are eliminated, while the visible objects are displayed by traversing their BSP trees.

A head-mounted display is used to view the environment in this system. Thus, two separate images with a slight offset must be produced for stereo display. This is done by the head-mounted display software.

2.7.4 Sound Production

Sound production is important to create an attractive and interesting art gallery. Because of the limitations of current hardware and software support, however, a minimal sound production facility is supported in this system. The basic idea is that when certain behaviors are executed, a set of the MIDI data required to generate the sounds in the current time step is produced and sent to the sound synthesizer attached to the system. The details of this are beyond the scope of this thesis.

2.8 Delivery Vehicle

The delivery vehicle consists of both a hardware system and a software system that are required for viewing and interacting with the environment. It is the front-end of the system for both single viewer and multiple viewers.

2.8.1 Delivery Vehicle Hardware

The hardware system of the delivery vehicle must have the following capabilities. It should allow the users to interact with RAG as if they were inside the gallery. it should also be able to display and update the visible images of the environment in real-time as the viewer's position changes from one location to another.

There are a number of input and output devices that can be used for the delivery vehicle. However, there is no fixed hardware configuration for the delivery vehicle since the configuration depends upon the art work involved, the budget for the display and the current hardware technology.

The DataGlove is the standard virtual reality input device. It can be used to determine the hand position and gesture. The Polhemus is another input device for determination of the head's position. It is reported that a new input device, the DataSuit, is being developed that covers the entire human frame in addition to the functionality of the DataGlove. Image processing technology can also be used as an input method. In this approach, several video cameras view the environment from different angles. The positions of the viewer's body parts can be determined using advanced image processing techniques. However, this approach is expensive in terms of hardware and software. In addition, it is hard to determine hand gestures due to its low sampling resolutions.

The output devices include the head-mounted display, EyePhone and several computers with at least moderate graphics capabilities. Another good, but expensive

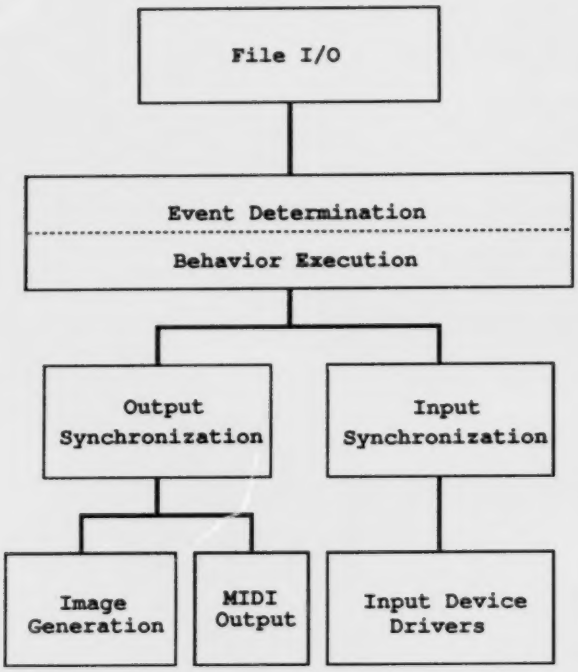

Figure 2: The delivery software.

alternative for the output device is a large screen installed on the walls of the room. Each wall gives a different view of the environment. In this approach, the participants (normally a few) will have a much better perception of the environment and they can easily move around to view the environment from different positions.

2.8.2 Delivery Vehicle Software

Delivery vehicle software is composed of a set of program modules which perform the following tasks: interfacing to the underlying operating system, calling the viewing program for event determination and behavior execution, synchronizing input and output devices, providing image display of the environment, and handling a wide range of input and output devices. Figure 2 shows a possible approach for configuring the delivery vehicle software. Logically, the delivery software shown in this configuration includes eight modules, each performing specific tasks.

The input driver module is the interface between the raw input device such as the DataGlove and the rest of the system. With this module, other parts of the system can be structured independently of the specific hardware used for the system. The delivery vehicle software only recognizes three types of logical input devices: hand, body and gesture regardless of the actual physical devices.

The input synchronization part is designed to simultaneously handle multiple input devices, and synchronize and repackage data from different input streams to satisfy expectations of other modules. Similarly, the output synchronization module synchronizes multiple output devices, especially for producing two images on two workstations required for a stereo display.

The file I/O module communicates with the underlying OS to store and retrieve data to and from the logical storage such as the UNIX file system.

Event determination and behavior execution have been discussed in the viewing program section. These modules are responsible for calling procedures in the viewing program when the input or output devices generate some events.

The sound production module takes care of sending MIDI data to the sound synthesizer attached to the delivery vehicle and producing synthesized sounds for the environment.

The image generation module is the most difficult part. It requires sophisticated graphics software and hardware support. If the hardware used in the art gallery supports the popular graphics capability like pipelining viewing transformation, 3D clipping, and polygon filling, the module may be easy to implement. If not, the software has to be powerful and fast enough to handle these tasks. The basic functions of the image generation module have been addressed in the viewing program section. The details of this module can be found in [Gre90] and are omitted from this thesis.

Chapter 3

Environment Editor and Its Implementation

As discussed in Chapter 2, the environment editor plays an important part in the RAG system. The environment editor in fact provides the artists with an easy-to-use design tool where the artists design and test(or preview) their art works: the environments. The term *environment* is sometimes used as a synonym for *artistic scenario of RAG* in the rest of this thesis. In addition to accomplishing the principal functions of the environment editor, the main aim of the user interface of the environment editor is to make it as friendly as possible.

In this chapter, we mainly concentrate on the implementation of the environment editor. As we go through the implementation, the concepts and techniques used in the environment editor are discussed in detail.

3.1 The Working Space

In this section, we discuss the setup of the working space where the artists, the input devices and the environments are placed. The term *working space* is used because the artists work interactively with the computer to create or display the environments in

this space.

3.1.1 Assumptions

Three assumptions have been made for the sake of simplification. The first assumption is that each environment uses a standard Cartesian coordinate system with the positive direction of the z axis pointing up. Thus, this assumption defines the orientation of the world coordinate system for all environments. This assumption does not have any impact on the range of environments that can be produced, but simplifies environment design and some interaction techniques, and makes it easier to share designs. The second assumption is that the ground plane of the working space happens to be the $z = 0$ plane in the coordinate system mentioned above. Again, this assumption reduces the difficulties in implementing some interaction techniques and makes it possible to share designs. The third assumption is that the z axis of the coordinate system for each space always points in the same direction as the z axis of the environment coordinate system. This assumption simplifies the space constraint algorithm which will be described in the next chapter.

3.1.2 Setup of the Working Space

The position and orientation of the hand in DataGlove are originally described in the device coordinate system. The room in which the user, the DataGlove and the computer reside has a rectangular shape and its own room coordinate system. One corner of the room is selected as the origin of the room coordinate system. This corner defines the x, y and z axes of the room coordinate system. The z axis of the room coordinate system points up, and the floor is the $z = 0$ plane. The x and y axes go along the two boundaries of the floor and meet at the corner. The mapping from device coordinate system to the room coordinate system can be viewed as a constant since the origin of the device, which is located at the Polhemus source of the

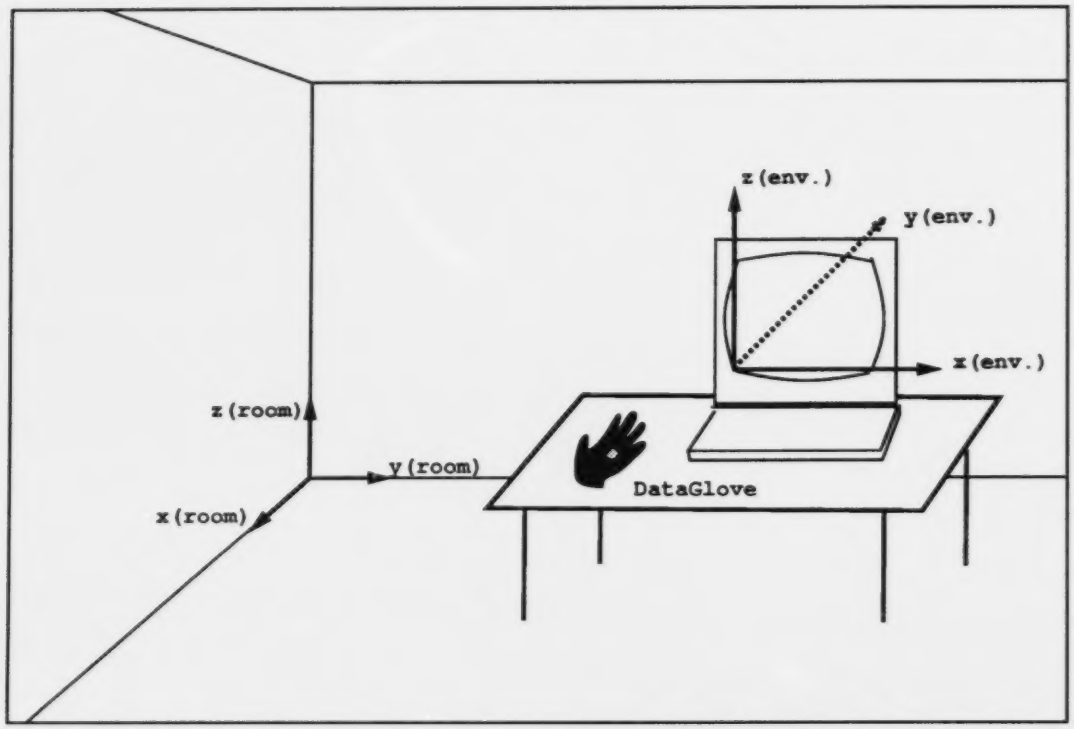

Figure 3: The setup of the environment coordinate system.

DataGlove, is fixed most of the time. The actual mapping is stored in the *.workspace* file that can be accessed by any application. The structure of this file and the tools available for workspace mapping are described in [Dep90b]. Finally, the room, the displayed objects and the DataGlove are described in the same working space by the environment coordinate system. The setup of the environment coordinate system is indicated in Figure 3.

The x axis of the environment coordinate system is defined to be parallel to the horizontal boundary of the display screen while the y axis is set to point inside the display screen. The z axis of the environment coordinate system points up vertically and the $z = 0$ plane resides on the floor of the room according to the two assumptions made in the previous section. The standard unit of measurement in the environment coordinate system is set to be 0.2 meter, while that of the room coordinate system is the meter.

3.2 Overview of Environment Editor Design

The environment editor performs a number of functions based on its role in the RAG system. The design principles for the editor are the friendliness of its graphical user interface, the consistency of the interfaces with other components, and the satisfaction of the requirements for the design of environments.

3.2.1 Functions of Environment Editor

The environment editor can logically be divided into the object editor, the space editor and the environment editing subsystem, which perform a number of operations on the following entities: object prototypes, object instances, masters, instances, spaces and the environment. These operations include creation, modification, deletion and storing of the entities.

As discussed in the previous chapter, the environment editor is used to create and modify the RAG scenarios that one or more users can interact with through the use of the delivery vehicle. Each environment is constructed from spaces and objects. Each object is made up from instances of masters. Each master is created from object instances and the existing masters. Each object instance is produced by parameterizing an object prototype. Therefore, creating and modifying an environment is composed of steps to create or modify spaces, objects, masters, instances, and object instances.

An environment consists of one or more spaces, each of which can contain any number of objects. A space serves as a unit in which the objects can be treated in a uniform way. The major tasks of the space editor are to construct spaces that have relatively simple structure [1], and to layout objects in these spaces.

The prototypes of objects defined by OML are the abstract descriptions of

[1] A space can be predefined as a special object prototype by OML. The geometric definition is the main concern of this kind of prototype.

objects that are relevant to the environment that the artists want to design. They are separately defined objects without being composed together to form new higher-level objects. A prototype may be thought of as defining a family of basic building blocks whose members vary in a few parameters. One of the main responsibilities of the environment editor is to instantiate the object prototypes and use them to construct more complex objects.

The environment editor uses the master-instance technique to create hierarchical objects. The object instances are used as building blocks to create higher-level entities, which in turn serve as building blocks for yet higher-level entities, and so on. This approach can save a lot of effort in designing the same kind of objects. The operations on masters are divided into two parts: one for creation and modification of object instances, and the other for creation and modification of masters.

After masters are instantiated and added into spaces, the remaining tasks are to define the dynamic aspects of the spaces and position the spaces in the environment to finalize the environment creation. As described before, each space defines a set of events, and each object has one or more behaviors that can respond to certain events that have occurred in the space. To control the dynamic behavior of objects, the editor provides a simple user interface to associate events with object behaviors. The user interface creates an event-behavior table for each space, that will be used by the viewing program. Adding spaces into the environment is the same as adding objects into spaces or masters. Techniques for these common operations are in fact shared by the object editing session, the space editing session and the environment editing session.

In addition to these basic functions, the editor also provides various ways to store, retrieve, and display the entities such as objects, spaces and environments. To assist viewing or manipulation of objects in the environment, some minimal 3D effects are added to the viewing section of the editor.

3.2.2 User Interface Layout of the Environment Editor

The environment editor has been implemented on a SGI Personal IRIS 4D in the UNIX environment. The screen layout of the environment editor is shown in Figure 4. The environment editor divides the display screen into the following five areas - button area, 3D viewing area, editing area, text input and output area and menu area, which are labeled as 1, 2, 3, 4 and 5 in Figure 4 respectively.

All the buttons used in the environment editor are uniformly placed in the button area. The corresponding action takes place when a button is pressed. There are three sets of buttons, which can be used to initialize the three input tools associated with the DataGlove, grids and potentiometers. There is also a group of buttons which serve as the editing commands in the environment, such as "Ok", "Exit" and "Redo".

The 3D viewing area is basically used as the editing and viewing window for the environment. A perspective projection is used to define the 3D viewing space. The environment editor allows the artists to adjust the viewing parameters from most parts of the environment editor in order to observe the environment from different angles.

The editing area assists with the editing and viewing of the entities in the environment. This area has two subareas. The left subarea is a window for selecting and viewing objects and spaces. The right subarea is reserved for potentiometers and 2D grids.

The text area is used for displaying help message and inputing text and commands.

The menu area is divided into an upper and lower subsection. The upper section is the environment editor menu system. The menu system consists of several submenus and is organized in a hierarchical structure. The menu system will be described in detail in later sections. The lower part of the menu area is used to

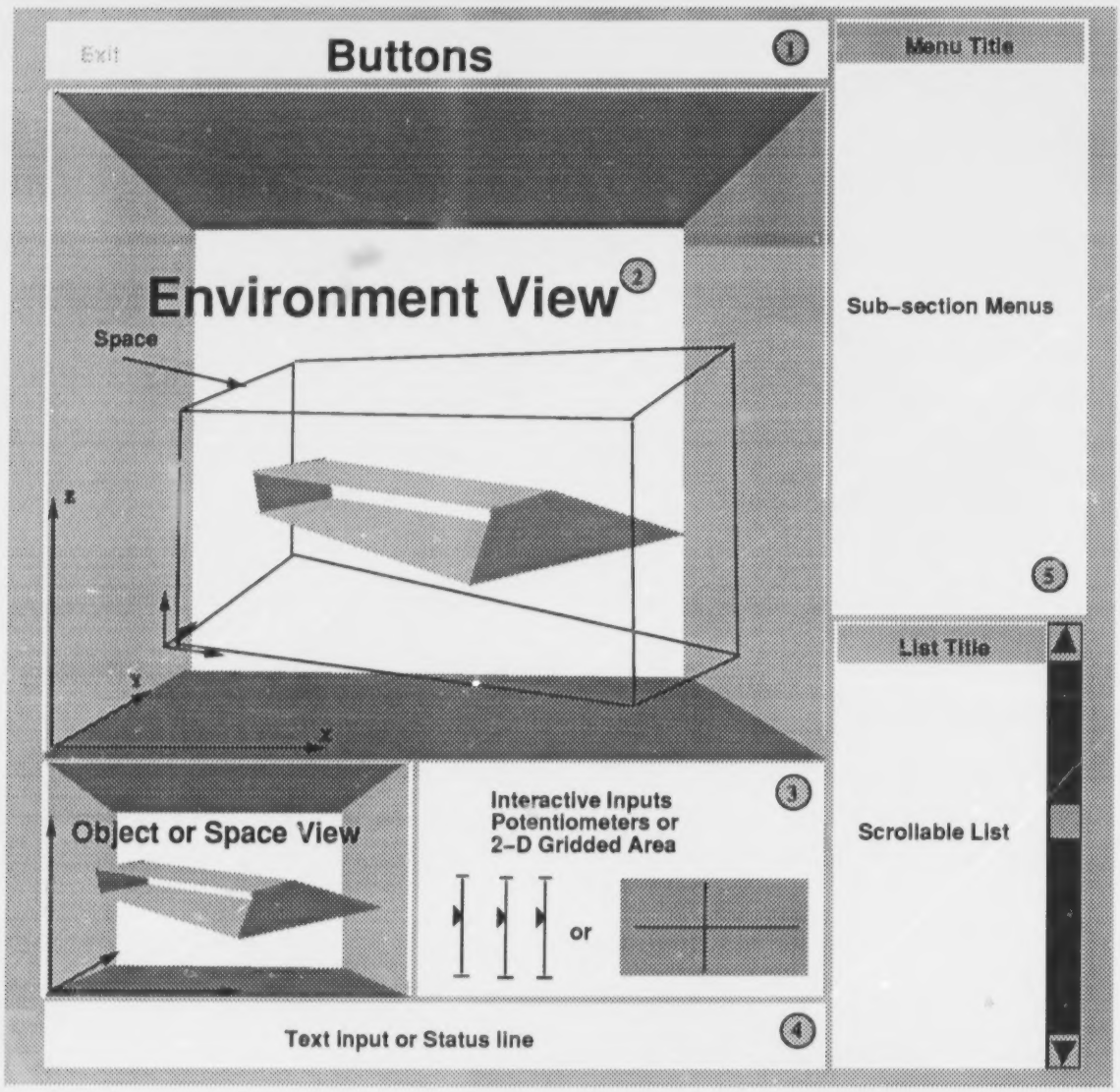

Figure 4: The layout of the environment editor.

display the names of all the types of entities in an environment database. Each list contains the names of the entities with the same type. For example, the list of all the masters in an environment database can be shown in this area.

3.3 Space Editor

Imagine that you are in an elevator on the top floor of a building. The building has the same number of floors as your present age. You push the "down" button and begin to descend slowly in time. You descend until reaching the year of your choice and stop the elevator. When the door opens, you enter a place that you can remember from that time. Examples of places which might be worth revisiting include: the house you grew up in, the school you first went to, and vacation places. After you arrive, take a fantasy walk through the place and interact with the things and people in the place that you used to be familiar with. You can choose one type of interaction activity, such as playing a hand-ball game with your old buddy in the school playground. To construct such a scenario, each of these places can be considered as a space. The concept of *space* enables the designer to group a set of entities and to associate messages, meanings and rules with them systematically. This section describes how a space is created through the environment editor.

The space editor is one of the three subsystems of the environment editor. It is used to construct and modify spaces. Specifically, the subsystem involves the following activities: space boundary editing, space content layout, and space modification.

3.3.1 Space Boundary Editing

Since a space has no visible representation and only serves as a container, its geometrical structure is relatively simple. This simple shape simplifies the implementation of spaces. Simple shaped spaces will simplify the procedure of space design, and speed

up event determination that the viewing program has to consider.

Technically, a space is defined by a collection of polygons. Like the definition of the geometry of other objects, the space boundary is input by defining a group of visible polylines although they are invisible when they are presented to the viewer. Therefore, the space boundary design is pretty straightforward.

The simplest way to define a space boundary is to use a predefined object prototype as the space boundary. With the editor, the artist only needs to instantiate the prototype with desired parameter values. Instantiating object prototypes is described in a later section.

Another way to construct a space uses a cross section parallel to the x-y plane plus a vertical profile, that is the minimum and maximum z values for the space. In this approach, the editor provides an editing area for the design and modification of the cross section. In this area, the artists enters a sequence of points of a 2-D convex polygon for the cross section, by using the rubber-band technique. Modifications to these points such as moving, deleting or adding a point into the existing polygon are also allowed. Two potentiometers are used to input the minimum and maximum z values. Spaces created by this way, therefore, are convex polyhedra[2] with cylindrical shapes.

3.3.2 Space Content Layout

A space becomes meaningful only if it contains one or more objects. The objects inside a space are called the *space content*. Thus space content layout is just the matter of laying out objects in the space.

As will be seen, an object in a space is actually an instance of a master that is created by the artist through the editor. Creating masters and the instances of a master is described in the subsequent sections. For now, we assume that a set of

[2] A convex polyhedron is defined as the convex hull of a finite set of points in 3D space.

masters have been created and their names are listed in the list area. To layout the space content the following steps are used:

- Select a master from the master list.
- Transform the master from the master coordinate system to the space coordinate system, thus an instance of the master(or an object) is created.
- Add the object to the space. Note that the boundary of the space is always displayed during these design stages.

The transformations from master coordinates to space coordinates can be any combination of rotation, scaling, and translation. There are two ways to input the transformation parameters. One way is to use a set of nine potentiometers that gives the values of a nine-tuple: (tx, ty, tz, sx, sy, sz, rx, ry, rz). tx, ty, tz are the distances of translation along the x, y and z axes, sx, sy, sz are the scaling factors along the x, y and z axes, and rx, ry, rz are the angles of rotation about the x, y and z axes. As done traditionally in computer graphics, a 4×4 matrix, which is the concatenation of translations, scalings and rotations, is used to represent the result of the transformation. The other way to input transformation parameters is to use the DataGlove. The DataGlove can smoothly rotate, resize and move the selected object, and place it within the boundary of the space. During each of the DataGlove operations, the corresponding matrix is formed and concatenated to the current transformation matrix.

3.3.3 Event-Behavior Table

The way to define the rules and meanings of a space is through the connection between a list of events associated with it and the behaviors of the objects within the space. An event occurrence is the cause of certain object behaviors, these behaviors are the

effect of the event. To perform specific behaviors when an event occurs, it is required to establish an event-behavior table that maps events into behaviors.

3.3.3.1 Structure of Event-Behavior Table

An event-behavior table is associated with each space. When an event occurs in a space, the system will look at the event-behavior table of that space and execute the behaviors corresponding to that event defined in this table.

Normally, the events in a space are linked as a list. So are the objects in the space. The behaviors associated with an object are also represented by a linked list. We refer to them as the *event-list*, *object-list* and *behavior-list* respectively. In order to refer to each item in a list, a code is assigned to each item. The code is determined by the position of the item relative to the first item of the list. For example, the first event in the event-list has code 0, the second has code 1, the third has code 2 and so on.

The event-behavior table is a 2-D array whose row and column correspond to events and objects, respectively, in the space. The item (i, j) in the array records the behavior of the object j that event i invokes. Initially, this array is set to Null(a behavior code that is never used such as 255). A Null value at (i, j) means that event i will not cause object j to do anything. Figure 5 illustrates the structure of the event-behavior table, where event 2 is mapped into behavior 3 associated with object 1.

3.3.3.2 User Interface to the Event-Behavior Table

The space editor submenu provides a menu entry for creating the event-behavior table for a pre-specified space. Once this menu item is selected, three lists: the event list of the space, the object list of the space and the behavior list of a selected object are displayed in the editing area. Each of the three lists has the first item as the

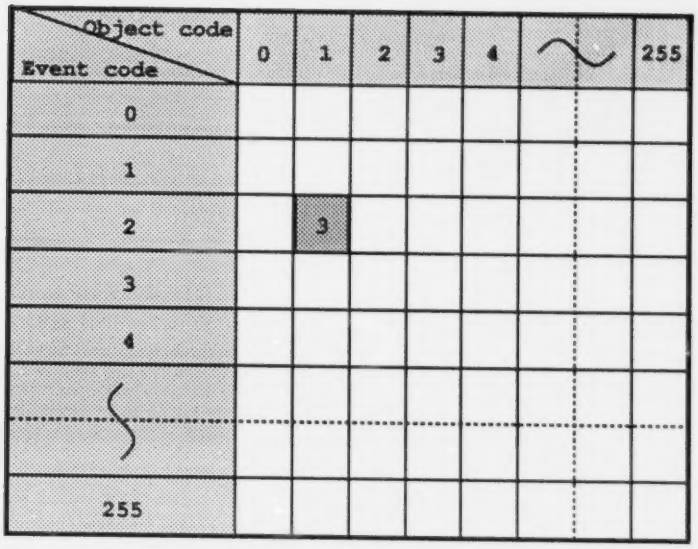

Figure 5: The event-behavior table.

default selection. After pressing the button "Associate", the user can re-select any combination from these three lists. The behavior list is, however, refreshed each time a new object is selected, to list the behaviors associated with the newly selected object.

When the selection is made, the "Ok" button is pressed in order to update the event-behavior table. This action simply sets the corresponding element in the event-behavior table to the newly selected behavior code. In addition to establishing the mapping, the user interface also allows the user to view the contents of this mapping. To do so, a button called "Display" is provided to enter the display mode.

In the display mode, as done above an event list is shown, and then a behavior-object list is displayed which contains a list of behavior names with their object names. These behaviors are those into which the event currently selected in the event list is mapped. The purpose of displaying the event-behavior table is to allow the user to check the mapping or modify the event-behavior table.

To maintain the correctness of the event-behavior table, the table needs to be updated properly when an object is deleted from a space. When this happens, all

the columns to the right of the column for the deleted object are shifted one position to the left. Addition of an object to a space does not affect the table as the new object is appended to the end of the object linked list. This is because the column corresponding to the new object in the event-behavior table is automatically set to Null.

3.3.4 Space Modification

Often, the designed space is not appropriate after it is placed in the environment. The space designer may want to make changes to the design. The environment editor provides the means to refine the space boundary, edit the space contents, modify the event-behavior table, and transform the whole space in the environment.

By selecting the name of the desired space from the space list, the artist can make changes to the space boundary. The way of accomplishing this modification is similar to the initial space design described in Section 3.3.1. Similarly, the artist is able to change the transformation of any object in the space. This can be done by using the DataGlove to translate, rotate or scale the object, or the potentiometers to reset the transformation parameter values. The procedure for modifying the event-behavior table is as follows. First, enter the event-behavior section, then select the event to be modified, and select the object that contains the behavior responding to the event. After the selections are done, the event-behavior table will be updated to reflect the new setting. Modification of the event-behavior table is more or less the same procedure as setting it up. Before making any changes, it may be helpful to view the event-behavior table by selecting the "Display" button.

3.4 Object Editor

The geometric models in the environment have a hierarchical structure. For example, a human figure is composed of the upper part and the lower part. The upper part is composed of two arms, upper body, neck and head. The lower part is composed of two legs and two feet. The object editor subsystem of the environment editor is used to construct this hierarchy. The object editor has the following functions.

1. Assign and edit the parameters of the object prototypes to create object instances.
2. Edit object instances.
3. Delete object instances.
4. Create masters from the object instances and existing masters.
5. Edit masters.
6. Delete masters.

3.4.1 Object Hierarchy in FDB

An environment created by the environment editor is organized in a hierarchical structure and stored in FDB. In FDB the entities of an environment are stored in frames. The hierarchical structure in FDB can be broken into two different levels. The high level structure consists of the environment frame, and a set of space frames that belong to the environment frame. The low level contains composite objects, masters, instances, object instances and primitives. The whole structure is represented by the frame hierarchy in FDB based on the GMP modeling schema [Dep89]. A new version of GMP has been implemented that follows the structure described in this section.

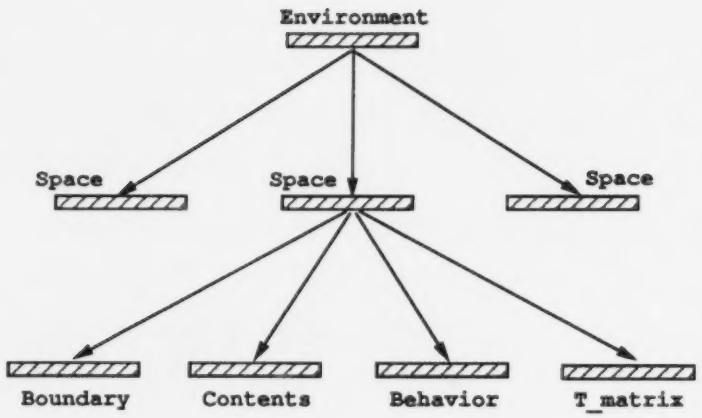

Figure 6: The high level structure of the environment database.

Figure 6 shows the high level frame structure of the environment. The root frame represents the environment. Its children are the frames representing the spaces in the environment. Each space frame has four key slots: Boundary, Behavior, Contents and T_matrix. The boundary slot points to a composite frame that describes the geometry of the space. The behavior slot contains the event-behavior table of the space. The contents slot is associated with another composite frame that has all the geometric information about the objects in the space. The T_matrix slot stores the transformation matrix from the space coordinates to the environment coordinates.

The low level structure, as shown in Figure 7, is rooted by a composite frame that has a collection of child frames, which point to subparts of the object. The child frame contains a transformation matrix and a pointer to one of the following frames: primitive, obj_inst, instance, or another composite frame. The transformation matrix in a child frame is used to transform the child object into its parent's coordinate system.

The primitive frame does not point to any other frame. It only points to a file that contains a collection of polygons stored in a standard format. The obj_inst frame contains the object prototype name and a pointer to a composite frame that defines the geometry for the object instance. Note that the composite frame is created

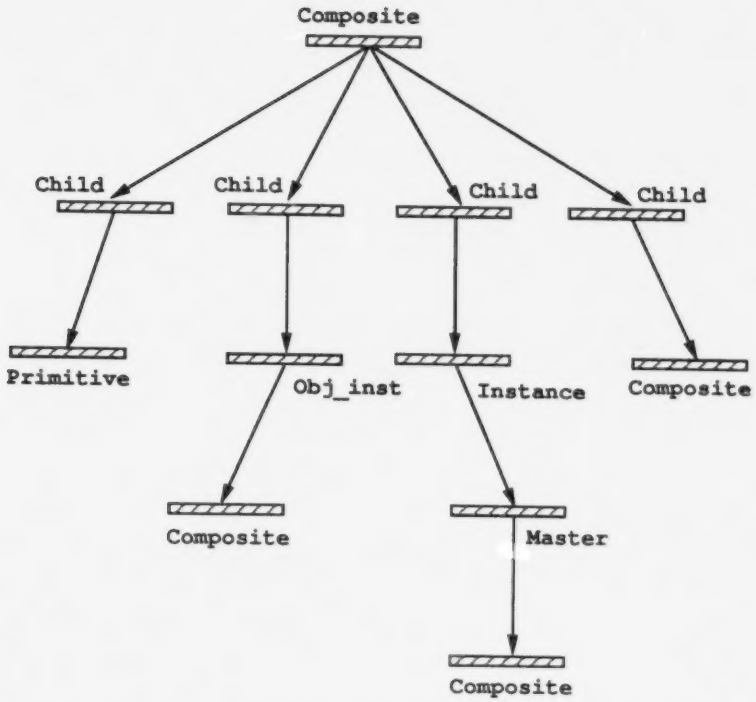

Figure 7: The low level structure of the environment database.

when an object instance is generated from its prototype. The instance frame has a slot pointing to the frame that stores the definition of its master. With the master-instance technique, the instance frame makes it possible to create a family of the same type of object with minimal storage. The transformation matrix for each instance is stored in the child frame. Like the object instance, the master frame points to a composite frame that is created in the master creation section. Thus, a master can have another master as its child. The creation of a master is described in the following subsections.

3.4.2 Generation of Object Instances

The input of the environment editor is a vocabulary of object prototypes with a set of routines that operate on these prototypes. The object prototypes and the routines are generated by the object compiler. The set of routines can also be considered as

part of the environment editor since the editor calls these routines to retrieve the information about the object prototypes.

Object instances, forming the bottom level of the object hierarchy, are created from object prototypes. The environment editor cannot edit the structure of an object prototype, but has the ability to assign any valid value to the parameters of an object prototype, and then regenerate the object. The regenerated object is considered as an object instance.

Object instance creation starts with selecting an object prototype from the list of prototypes. Once the prototype has been identified, its parameters are listed on the screen. The routine *OML_get_parameters()* is used to retrieve all the parameters of the given object prototype. Based on the types of the parameters, several interactive techniques are used to input the values for these parameters. The current implementation of the environment editor allows input of the following six parameter types: integer, floating point number, 2D point, 3D point, 2D polyline and 3D polyline. The detailed methods for inputing these types of parameters is described in Chapter 4, which is dedicated to a number of important techniques used in the environment editor. After the parameter values are obtained from the user, the routine *OML_parameter_value()* is called to set the given parameters to their corresponding values. Then the user is asked to input a name for the instance and the routine *OML_interpreter()* is called to create an obj_inst frame in FDB and produce polygons that describe the geometry of the object instance. As described above, all the geometry information for the object instance can be reached through the obj_inst frame.

The system keeps an object list that contains all the object instances having been created. As requested, the editor can show the object list in the list area. To modify an existing object instance, first, the instance needs to be identified from the object list, second, the object is displayed, and then the normal section for inputing

parameter values is activated. From there, the user can change the parameters to regenerate the object instance.

3.4.3 Master Editing and Instance Creation

A master is a composite object that has been assigned a name by the designer. The composite object is composed from a collection of the object instances or the existing masters. A master has its own master coordinate system. Creating a master is equivalent to placing a collection of object instances or existing masters in the master coordinate system.

Master creation starts with selecting the type of object to be added: object instance or master. The environment editor shows either the object instance list or the existing master list according to the designer's choice. Then a set of potentiometers or the DataGlove can be used to establish the transformations from the selected object's coordinates to the master coordinates. Once an object instance is in-place, the designer can add another object to the master using the same procedures. The added objects can be viewed as the children of the master. The name of the new master is added to the master list and can be used to create other masters.

Internally, when master creation starts, a master frame is created. Since the master frame points to a composite frame, these in-place objects are linked together as the children of the composite frame. The transformation matrix of each object is recorded in the corresponding child frame of the composite frame. If an object is created from one of the existing masters, an instance frame is created. The instance of the pre-defined master is added to the new master.

Modification of an existing master includes deletion of objects from the master, addition of new objects to the master, and rearranging the objects in the master.

To delete an object in a master, first we need to identify the object to be deleted either by selecting it from the object list of the master or by grabbing it using

the DataGlove. The deletion takes place when the designer confirms the deletion operation by entering "yes" from the text area. The other two operations of modifying a master are similar to the methods used in creating a master.

3.5 Environment Editing Subsystem

As shown in Figure 6, the environment is composed of a number of spaces, each of which contains a set of objects. The key operation of the environment editing subsystem is to place the created spaces in the environment coordinate system or the world coordinate system. Since a space as well as the objects in the space must be created before a space is placed in the environment, this subsystem is the highest level operation. It can be regarded as the super-subsystem of the other two editors: the space editor and the object editor.

The functions of the environment editing subsystem include the following.

1. Creating a new environment:
   - creating spaces by calling the space editor module.
   - adding spaces to the environment.
2. Editing existing environments.
   - adding new spaces into an environment.
   - modifying spaces in an environment.
   - removing spaces from an environment.
   - editing the event-behavior table of a space in an environment.
3. Saving an environment to the disk.
4. Retrieving an environment from the disk.
5. Viewing the complete environments.

3.5.1 Environment Creation and Modification

The environment editor provides a list of the spaces in an environment in the scrollable list area of the screen. The user can select a space from the space list. The selected space is then added into the environment by using the DataGlove or a set of potentiometers for entering transformation parameters. The space coordinate system is transformed to the environment coordinate system when the selected space is added to the environment. The transformation matrix is stored in the t_matrix slot of the space frame. These operations for transforming spaces to the environment are the same as those for transforming objects into a master, or transforming instances into a space.

Again, the environment editing subsystem also allows the user to remodel a created environment. New spaces can be added into an existing environment by selecting a space from the space list. Any space in an existing environment can be deleted also by selecting it from the space list. The event-behavior table for each space in the environment can be modified. In addition to the modification of spaces, the subsystem also allows the modification, addition and deletion of the objects inside a space. This is done by revisiting the space editor module and the object editor module.

3.5.2 Other Operations

Several menus provide other functions that are used in environment editing. They are "Save", "Delete" and "View". When the "Save" item is selected, the environment being created or modified is stored in a FDB database and the old version of the environment is overwritten. When "Delete" is selected, the user is asked to confirm whether to delete the environment from the database or not. If so, the environment is deleted from the disk. The "View" menu item can be selected to view the desired environment from different angles by adjusting viewing parameters.

Chapter 4

Manipulation of 3D Objects

One of the objectives for the environment editor is user friendliness. Advanced interaction techniques are essential in order to achieve this goal. In addition to accomplishing the basic functions of the environment editor, this thesis has also explored several interaction techniques for manipulating 3D objects in the environment editor. These methods vary from simple but useful scrollable lists to directly interacting with 3D objects using the DataGlove. In this chapter, these interaction techniques and the algorithms for manipulating 3D objects are discussed in detail.

4.1 Methods for Editing Objects

The environment editor provides the artist with a friendly interface to create and edit the RAG scenarios. The following six types of interaction techniques: DataGlove, grids, potentiometers, scrollable lists, buttons and hierarchical menus are used to invoke commands, select entities, edit parameters of object prototypes, perform transformations, and so on.

4.1.1 DataGlove

One of the important features of the environment editor is the use of the DataGlove as an input device. The DataGlove is suitable for entering values of three dimensional parameter types in object creation and setting transformations for objects and spaces. It is also convenient for sending commands to the editor using various gestures. The virtue of employing the DataGlove as an input device is that it provides a more natural and efficient way to manipulate the objects in the 3D environment than the conventional 2D devices such as the mouse.

The DataGlove software package built at University of Alberta provides two kinds of mechanisms: gesture recognition, and position and orientation determination of the DataGlove. With the DataGlove software package, the environment editor is able to implement the direct interaction paradigms by using DataGlove. The direct interaction paradigms using the DataGlove are described in the following subsections.

4.1.1.1 Editing Path3 Parameters

A path3 type is a sequence of points in a 3D space, with adjacent pairs of points connected by straight lines. To view the 3D polylines, three small windows are set up in the editing area on the screen. The window on the left is used to display the 3D polylines using a perspective view. It is called the display window. The middle window and the window on the right are used to display parallel projections of the 3D polylines onto the x-y and x-z planes, respectively. They are called x-y and x-z projection planes. The user directly manipulates the 3-D points of the polyline in the left window. A rubber-band technique has been employed to draw a line between the previously input point and the current position of the DataGlove. The other two windows are designed to make interactive input easy. With the two projection windows, the user has a better view of the 3D polyline, and thus can accurately position the input points.

To use the DataGlove to enter a 3D polyline, a set of gestures is used, including *enter-point*, *end-input*, *grab-point*, and *move-path3* gestures. When the DataGlove makes an entering-point gesture, the current position of the DataGlove is entered as one point of the path3. A sequence of these gestures defines a path3. The last point of the path3 is entered by making the end-input gesture. When a point is entered, a new line is drawn in the display window. At the same time, the x-y and x-z projections of the line are drawn in the corresponding projection windows. Each of the two projection windows has a grid in the background, which is helpful for editing the polyline.

The environment editor allows the user to modify an existing path3 by relocating one point or the whole polyline. If the hand is close to a point of the polyline and the grab-point gesture is made, the point can be moved to a new position as the hand is moved. The end-input gesture is used to finalize the new position. The whole polyline can be moved by making a *move-path3* gesture. In order to grab a desired point, the two projection windows can be used to help the user to move his/her hand close enough to the point.

4.1.1.2 Setting Transformations

Another important use of the DataGlove is to establish and edit the transformations from the object coordinate system to the master coordinate system when an object is added to a master, from the master coordinate system to the space coordinate system when a master is added to a space, and from the space coordinate system to the environment coordinate system when a space is added to an environment. The transformation can be broken down into three separate transformations: translation, rotation and scaling along x, y and z axes. There is a predefined gesture for each of these three transformations.

When a translation gesture is made, the object currently grabbed by the

DataGlove[1] starts moving in the direction of the movement of the DataGlove. The distance which the object moves is proportional to the distance the DataGlove moves. Two constants, *TRAN_TOL* and *tfactor*, are used to determine when and how to move an object. The following formulas are adopted in the system.

$$if(|\Delta x| \geq TRAN_TOL), \quad tx = tx + \Delta x \times tfactor;$$

$$if(|\Delta y| \geq TRAN_TOL) \quad ty = ty + \Delta y \times tfactor;$$

$$if(|\Delta z| \geq TRAN_TOL) \quad tz = tz + \Delta z \times tfactor.$$

From the above formulas, the translation operation takes place when the DataGlove's movement is greater than the tolerance *TRAN_TOL*. The Δx, Δy and Δz are the displacements of the DataGlove along x, y and z axes. The *tfactor* determines the actual distance of the object's motion. The values of *TRAN_TOL* and *tfactor* can be changed on the fly by the user. Note that the DataGlove uses the world coordinate system.

For scaling, if the DataGlove moves along the positive x direction in the world coordinate system, the scaling parameter sx increases at a constant speed. If the DataGlove moves along the negative x direction, the scaling parameter sx decreases at a constant speed. An object can also be scaled in y and z directions, individually. If the DataGlove moves in an arbitrary direction, the x, y, z components are calculated individually, and the scaling is executed sequentially along each of x, y, z directions.

$$if(\Delta x \geq SCALE_TOL) \quad sx = sx + sfactor;$$

$$if(\Delta x \leq -SCALE_TOL) \quad sx = sx - sfactor;$$

$$if(\Delta y \geq SCALE_TOL) \quad sy = sy + sfactor;$$

$$if(\Delta y \leq -SCALE_TOL) \quad sy = sy - sfactor;$$

[1] For any transformation gesture, a grabbing gesture must be made first to identify the entity to be transformed.

$$if(\Delta z \geq SCALE_TOL) \qquad sz = sz + sfactor;$$

$$if(\Delta z \leq -SCALE_TOL) \qquad sz = sz - sfactor.$$

The scaling operation takes place when the DataGlove's movement is greater than a scaling tolerance *SCAL_TOL*. The constant *sfactor* is used to control the speed of the scaling. Again, the values of these two constants can be adjusted by the user in the environment editor.

Similarly, the rotation by the DataGlove takes place when the DataGlove's movement exceeds a given tolerance *ROT_TOL*. The displacements of the DataGlove in x, y, z directions are tested against *ROT_TOL*. The angle of the rotation along the x, y, z axes is the constant *rfactor* for each valid rotation gesture using the DataGlove.

$$if(\Delta x \geq ROT_TOL) \qquad rx = rx + rfactor;$$

$$if(\Delta x \leq -ROT_TOL) \qquad rx = rx - rfactor;$$

$$if(\Delta y \geq ROT_TOL) \qquad ry = ry + rfactor;$$

$$if(\Delta y \leq -ROT_TOL) \qquad ry = ry - rfactor;$$

$$if(\Delta z \geq ROT_TOL) \qquad rz = rz + rfactor;$$

$$if(\Delta z \leq -ROT_TOL) \qquad rz = rz - rfactor.$$

4.1.2 Grids and Potentiometers

Besides the DataGlove, grids and potentiometers are two other input techniques used in many places of the environment editor. A potentiometer is basically used to input any numerical type data such as angles of rotation and scaling factors. A grid is usually used to input 2D polygons or polylines.

The grid package is independent of the environment editor. The functions in this package can be used in other software running on the IRIS. To satisfy the need of the environment editor, this package provides the following features:

- The units of x axis and y axis are set up dynamically.
- The intervals along the x and y axes of the grid are adjustable.
- The color intensity of the grid is changeable.
- Input points can be snapped to the closest grid intersection. The snapping function can be turned on or off.
- A rubber-band technique is used to define polygons or polylines.

The user starts entering a polygon by pressing the left button of the mouse. A rubber-band line is drawn between the previous input point and the current cursor position. When the left button is pressed again, the rubber-band line becomes one of the edges of the polygon, and is statically drawn on the screen. When the user presses the right button, the polygon is closed by the edge between the last input point and the first input point. Similarly, a polyline can be entered in this way. In entering points, if the snapping function is turned on, each point to be entered is snapped to the nearest grid intersection. With this function, the user can precisely position a point.

The potentiometers package is also independent of the environment editor. It mainly provides the following features.

- The type of values of the potentiometers can be either integer or float.
- The domain of the potentiometers can be set up interactively.
- The interval of the potentiometer values is adjustable.
- The intensity of the potentiometer color is changeable.
- Snapping can be turned on or off. With the snapping on, the indicator jumps at the specified intervals.

The potentiometer has been used in the following places.

- The transformation matrix can be set by using a bank of 9 potentiometers to specify x, y, and z components of each of the basic transformations: rotation, scaling and translation. The approach of using potentiometers enables the user to define the transformations more precisely than using the DataGlove.
- The viewing parameters are adjustable in most parts of the environment editor by using a set of potentiometers.
- point2[2] and point3[3] type parameters are entered using potentiometers.
- The settings of the DataGlove transformation constants (refer to the previous section) uses a set of potentiometers.

4.1.3 Buttons and Scrollable Lists

There are many uses of buttons in the environment editor. A button serves as an action trigger. When it is pressed, a certain action takes place, or a certain mode is activated. The implementation of the button program is straightforward and can be re-used in other software systems.

The scrollable list package is another portable program. It is implemented on top of the existing menu package with some modifications. The menu is static, that is, all menu items are added to the menu and the layout is then calculated, while the scrollable list is dynamic in the sense that an item can be added at any time and the layout is calculated dynamically whenever a new item is added. When the number of items exceeds the height of the list area, a scroll bar is added beside the list. With the scroll bar, the user can scroll the list items up or down by pressing the arrows located at the top and the bottom of the bar, respectively. An indicator is used to indicate

[2]point2 parameter is defined as a point in 2D space.
[3]point3 parameter is defined as a point in 3D space.

which portion of the list is visible at the current time. The selection mechanism in the scrollable list is the same as the menu.

The scrollable list programs have the following uses. As described in Chapter 3, each environment created by the environment editor has one space list, one master list and one object prototype list. Each space in the environment also has its own object list. To present these lists to the user, the environment editor displays them in the list area in which the user simply selects an item and the item is highlighted to indicate the selection. In addition, the scrollable list is also used to list the created environments from which the user can select to edit the selected environment.

4.1.4 DataGlove vs. Potentiometers

Both the DataGlove and potentiometers can be used to perform transformations of objects in the environment. However, they have different features in terms of the motion smoothness and the preciseness of the end condition. Using the DataGlove, the objects being transformed go through intermediate configurations in real-time until they reach their final destination. The DataGlove provides smooth transformation. But, the objects jump to their new location without going through intermediate configurations when a set of potentiometers is used. On the other hand, the transformations can only be terminated on an aesthetic pleasing end condition by using DataGlove. The user's hand in DataGlove stops gesturing when the object's position looks good to him/her. However, the potentiometers enable the user to enter precise end conditions.

4.2 Grabbing Operations

Selecting a desired object in the environment is the first step in manipulating objects. The DataGlove can be used to select an object by issuing the grabbing gesture. Thus,

the user can pick up the virtual objects in a natural way as he does with real objects. In this section, we discuss the grabbing mechanism in the environment editor.

4.2.1 The Problems

When the user attempts to grab an object in the environment using the DataGlove, two conditions must be satisfied. First, the DataGlove must be close enough to the object to be grabbed. Second, a grabbing gesture has to be made before the grabbing operation takes place. Once an object is grabbed, the environment editor should report that an object is selected and the grabbed object should be labeled as a selected object.

In order for the user to position his hand in the environment, an articulated hand that spatially corresponds with the user's real hand, and is directly controlled by the DataGlove is displayed on the screen. With the articulated hand, the user is able to move his hand towards the desired object and decide whether the grabbing gesture needs to be issued or not. If the grabbing gesture is made and no object is close enough to be grabbed by the DataGlove, nothing will happen. When the desired object is indeed grabbed, the selected object is flashed.

There are several problems with grabbing operations.

- How to decide whether an object is close enough to be grabbed by DataGlove.
- How to measure the distance between the DataGlove and the object.
- How to distinguish the desired object from crowded objects.

The DataGlove provides a natural and direct means to select objects. However, when the desired object is obscured by other objects or objects are close to it, the DataGlove can fail to identify the object. In order to solve this problem, we provide the user with an object name list from which the user can select an object by picking its name.

4.2.2 The Grabbing Algorithm

As mentioned above, we assume that an object is grabbed if the position of the DataGlove is close enough to the object and the grabbing gesture has been made. Thus the environment editor needs to measure the distance between the DataGlove and the object and detect the grabbing gesture.

An environment created by the environment editor consists of a set of spaces, each of which is composed of a set of objects. Each object is an instance of a master. A master is a composite object with a name. A composite object is made from other composites, instances of masters, object instances, and primitives. The whole environment is composed of an object hierarchy tree of which the leaves are primitives. By traversing the tree, we eventually get a collection of primitives, each being a collection of polygons. Without losing generality, therefore, we make an assumption that an environment created by the environment editor is composed of a collection of polygons, that is, the geometry of each object in the environment is actually shaped by a collection of polygons.

The pseudo code for the grabbing algorithm is shown in Figure 8. Basically, the grabbing algorithm looks for the first polygon that is close enough to the DataGlove position by traversing the object hierarchy tree. In the object hierarchy tree, the leaves of the tree are primitives. The function *grabbing_an_object* recursively searches through masters, instances, composites, child frames, and object instances until a primitive is found. If the primitive is what is expected, the recursive procedure stops; otherwise, the function continues on to other branches of the tree. The function *map_to_object* is responsible for mapping a frame retrieved from the FDB database to the object data structure in the environment.

The high level structure of an environment is stored in the FDB database. The primitives are stored in a separate file. The file name and the entry tag of each primitive in the primitive file are, however, stored in the primitive frame in the FDB.

```
int grabbing_an_object( glove, obj, Tmat)
Hand        glove;
Object      obj;
Trans_matrix  Tmat;
{
        int             obj_type, fr, tag;
        Object          child;
        Trans_matrix    t, Tmat0;
        obj_type = getvalue( obj->db, obj->frame, Fheader );
        switch  ( obj_type )    {
                case    ( master ):
                        fr = getvalue( obj->db, obj->frame, Definition);
                        child = map_to_object( fr, obj->db );
                        return  ( grabbing_an_object( glove, child, Tmat ) );
                case    ( instance ):
                        fr = getvalue( obj->db, obj->frame, Master_frame);
                        child = map_to_object( fr, obj->db );
                        return  ( grabbing_an_object( glove, child, Tmat ) );
                case    ( composite ):
                        fr = getvalue( obj->db, obj->frame, Child_list_start );
                        Tmat0 = Tmat;
                        while   ( fr != -1 )    {
                                child = map_to_object( fr, obj->db );
                                if  ( grabbing_an_object( glove, child, Tmat ) == GRABBED )
                                    return      GRABBED;
                                else{
                                    fr = getvalue( child->db, child->frame, Child_list_next );
                                    Tmat = Tmat0;
                                }
                        }
                        return      UNGRABBED;
                case    ( child ):
                        t = getvalue( obj->db, obj->frame, T_matrix );
                        Tmat = t * Tmat;
                        fr = getvalue( obj->db, obj->frame, Definition );
                        child = map_to_object( fr, obj->db );
                        return  ( grabbing_an_object (glove, child, Tmat ));
                case    ( object instance ):
                        fr = getvalue( obj->db, obj->frame, Definition );
                        child = map_to_object ( fr, obj->db );
                        return  ( grabbing_an_object ( glove, child, Tmat ));
                case    ( primitive ):
                        tag = getvalue( obj->db, obj->frame, Prim_tag );
                        return  ( grabbing_a_primitive( glove, tag, Tmat ) );
                defaults:
                        return      UNGRABBED;
        }
}
```

Figure 8: The grabbing algorithm.

To handle the low level modeling details, the polygon modeling package developed at University of Alberta is used. The polygons of the primitive are retrieved in the function *grabbing_a_primitive*. Then, the function *grabbing_a_primitive* calculates the distance between the DataGlove and the primitive to determine whether the primitive is grabbed or not.

Given the DataGlove position G, and the set of edges E that define an object, the distance between the DataGlove and the object is defined as

$$d(\bar{e}) = \min_{e \in E} d(e).$$

The distance between the DataGlove position G and an edge e of the object is defined as the Euclidean distance between the DataGlove and the closest point on the edge.

$$d(e) = min\{|p - G| : p \in e\}.$$

The edge is defined in the parametric form, $P(t) = P_0 + t(P_1 - P_0)$ where $(0 \leq t \leq 1)$, P_0 and P_1 are the end points of the edge. According to geometric theory, the closest point P_{min} will occur at either one of the two end points of the edge or the intersection of the edge and its perpendicular line passing through G, if the intersection exists. $P(t)$ is the intersection point when $(G - P(t)) \cdot v = 0$ where $v = P_1 - P_0$. Expanding this, we get

$$(G - P_0 - tv) \cdot v = 0,$$

$$(G - P_0) \cdot v = tv \cdot v,$$

$$t = \frac{(G - P_0) \cdot v}{v \cdot v}.$$

Plugging t into the function for the edge, we obtain the point closest to G, $P_{min} = P_0 + \frac{(G-P_0)\cdot v}{v \cdot v} v$. If t is within [0, 1], that means $P_{min} \in P$. Thus, the distance between P_{min} and G is the distance that we want. If t is not in [0, 1], P_{min} is on the extension of the edge. In this case, the distances from G to the end points of the

edge, P_0 and P_1, are calculated. Suppose that the distances obtained are d_1 and d_2, respectively. Thus the distance we want to obtain is equal to the minimum value of d_1 and d_2.

4.3 Space Boundary Constraints

The concept of space introduced into the environment editor has two purposes: to group a set of objects as the content of a space, so that they can be manipulated as one unit and assigned rules and meanings by the events-behavior table; and to restrict the objects in the space to lie completely within the space. The space boundary constraint means that the objects in a space are not allowed to move out of the space boundary. The space boundary constraint is applied when an object is added into a space or when an object is moved, scaled and rotated in a space.

The space constraint algorithm is developed for the following two tasks:

- An object is tested for containment within its space boundary when it is added into the space and when it is transformed in the space.
- Performing the transformation on the objects with the restriction that the objects always stay within the space boundary.

The space constraint algorithm described below is based on the following assumptions. All the primitives that define an object are convex polyhedrons. The z axis of the space coordinate system always points up according to the assumption made in Section 3.3. A space boundary is a convex polyhedron taking a cylindrical shape with the cross section parallel to the x-y plane being a convex polygon. Thus, the first task of the algorithm is to determine whether all the vertices of an object are inside the space boundary. If so, the object is in the interior of the space boundary.

To determine if a vertex (x, y, z) is inside a space requires two steps. First, the z value should be greater than the minimum z of the space and less than the

maximum z of the space. Second, the (x, y) should be inside the cross section of the space that is parallel to the x-y plane.

Given a simple polygon P and a point Q, the algorithm for determining whether the point Q is in the interior or the exterior of P is based on the following lemma.

Lemma. Let l be a horizontal ray that starts from Q to the right of the point. Let k be the number of intersections of l with the boundary of P. Then k is odd if Q is in the interior of P; k is even if Q is in exterior of P.

The proof of the above lemma can be found in [Prep85].

The second task of the space constraint algorithm is to modify the transformation that will be applied to the object if that transformation would move the object outside of the space boundary. The transformation of an object is the concatenation of a sequence of rotations, scalings or translations along the x, y and z axes. The transformation end conditions are represented by a nine-tuple $(tx, ty, tz, sx, sy, sz, rx, ry, rz)$. The transformation corresponding to each element in the nine-tuple is executed individually. Let e_0 be one of the element in the nine-tuple $(tx, ty, tz, sx, sy, sz, rx, ry, rz)$, $T(e_0)$ be the transformation matrix with the end condition e_0, V be the point set defining the space region, and P be the point set defining an object, $P \subset V$. We define $\delta(e_0)$ as

$$\delta(e_0) = \begin{cases} max\{\ e\ |P \times T(e) \subset V, \quad 0 \leq e \leq e_0\} & \text{if } e_0 > 0 \\ min\{\ e\ |P \times T(e) \subset V, \quad 0 \geq e \geq e_0\} & \text{if } e_0 < 0 \end{cases}$$

According to the above definition, $\delta(e_0)$ is equal to e_0 if the object P is still in the interior of the space boundary V after the transformation with the end condition e_0 applied to the object. Otherwise, $\delta(e_0)$ is an extreme value in the interval of e_0 and 0, such that by the transformation $T(\delta(e_0))$, the object is transformed to a position where it is still in the interior of the space boundary, yet attaches to the space boundary. We use the *transformation bisection method* to find $\delta(e_0)$ shown below.

```
function δ(ε0): real
begin
    if (P × T(ε0) ⊂ V)    return(ε0);
    else   begin
            A := 0;  B := ε0;  M := (A+B)/2;
            while (|A − B| > ε)
            begin   if (P × T(M) ⊂ V)     A := M;
                    else     B := M;
                    M := (A+B)/2;
            end
            return(A);
    end
end
```

Suppose the object in a space is to be transformed with end condition ϵ_0, $\epsilon_0 > 0$. If the object is still in the interior of the space boundary after the transformation, then $\delta(\epsilon_0) = \epsilon_0$. Otherwise, the *transformation bisection method* looks for $\delta(\epsilon_0)$. Suppose we have an interval $[0, \epsilon_0]$, and the interval has such a property that the object is in the interior of the space boundary after being transformed with end condition 0, and that the object is not in the interior of the space boundary after being transformed with end condition ϵ_0. We look at the midpoint of the interval. If the object is in the interior of the space boundary after transformed with the end condition $\frac{\epsilon_0}{2}$, then we know $\delta(\epsilon_0)$ lies in the interval $[\frac{\epsilon_0}{2}, \epsilon_0]$. Otherwise, $\delta(\epsilon_0)$ lies in the interval $[0, \frac{\epsilon_0}{2}]$.

Now we have a new interval (either $[0, \frac{\epsilon_0}{2}]$ or $[\frac{\epsilon_0}{2}, \epsilon_0]$ depending on whether the object is in the interior of the space boundary after being transformed with end condition ϵ_0) that has the same property as the original interval. So we apply the same procedure to the new interval. We can repeat this sequence of steps, each time

narrowing the possible interval in which $\delta(e_0)$ must lie. Eventually the size of the interval will become so small that we can treat it effectively as zero and say that the proximity of $\delta(e_0)$ lies at the point that results.

The above transformation bisection method applies to each element of the nine-tuple $(tx, ty, tz, sx, sy, sz, rx, ry, rz)$ when an object within a space is transformed.

Chapter 5

Conclusions and Future Work

This thesis has examined the issues related to virtual reality as an artistic medium. RAG, which is a recently proposed artistic virtual reality system, has been introduced to explore methods and tools for building virtual reality applications in arts. The environment editor, which is one of the components of the system, has been implemented.

The environment editor uses a set of object prototypes defined by the object modeling language, which is used to define the geometry and behavior of objects appearing in virtual environments, as its input. The environment editor enables the artists to set up the parameters of object prototypes, instantiate object instances, construct objects and spaces, as well as build environments. The objects in an environment are organized in a hierarchical manner. A set of routines are provided to deal with the hierarchical structuring of objects, master-instance relationship and transformation in a virtual environment. The environment editor provides a FDB framework into which virtual environments can be stored. Other components of RAG thus can retrieve the environments from the FDB database.

In addition to providing the above basic functionalities, the environment editor also accommodates a rich set of building blocks for graphical user interfaces.

Scrollable lists, buttons, grids, potentiometers and menu systems are all useful to many kinds of graphical applications. In the environment editor, they are the critical input/output means for the user to communicate with the editor.

The DataGlove is used in the environment editor as an interactive device for directly manipulating the objects in a virtual environment. Direct manipulation of objects is the key to a successful design of a piece of art work. It has two advantages. First, direct manipulation of objects makes it more efficient to design 3D scenes than two dimensional tools like potentiometers. Second, it gives the designer (the artist) a feeling of being inside the environment. In this way, the artist feels less that he is playing with a computer tool but more that he is creating art work.

In this environment editor, the DataGlove is used to control the three axes concurrently in order to apply transformations to objects. It is also used to grab objects. However, when using the DataGlove to grab an object in space, the grabbed object does not appear to be trapped by the hand in the DataGlove in a natural, physically credible way. The user's hand does not have force feedback and tactile feedback. In an ideal approach to grabbing a virtual object by DataGlove, the user would have visual and haptic feedback of grabbing something in his/her hand as he/she does in reality.

Simulating the geometric deformation of the hand during a grabbing action is a very complex problem [RG91]. Some progress on this topic has been made recently [JPGT89] [Iwa90] [RG91]. To animate hand grabbing provides a challenging topic for future improvement of the environment editor.

To achieve force feedback and tactile feedback effects, scientists and engineers are exploring methods or devices for creating force and tactile feedback. A new version of the DataGlove with force and tactile feedback has recently appeared. Once the technology for this kind of DataGlove is mature, the environment editor using the new DataGlove facilities as input means will be more realistic and effective.

While visual and haptic feedback is one problem for using the DataGlove in the environment editor, conveying depth information of 3D scenes to the viewer is another complicated problem. The current implementation of the environment editor provides limited support for depth perception. Hidden surfaces are removed by the Z-buffer method. Constant shading with pseudo colors is added on the surfaces of objects. It is obvious that these two simple methods are insufficient. In fact, it is almost impossible to judge how far an object is away from another object in environment editor. In particular, it is quite difficult to place an object precisely in a 3D scene most of the time. To improve depth perception, the proposals are outlined in the following paragraph.

Rendering the objects using a lighting model can help to determine the distances between objects. If the shadows of objects are modeled and they are updated according to the motion of the objects, the user can easily place or move an object to the desired locations. Apparently, this approach substantially increases interactive response time.

In performing tasks such as placing an object precisely in a 3D scene, the 3D snap-dragging techniques [Bie90] can be used. The 3D snap-dragging uses a gravity function, a 3D cursor and a set of alignment objects. The gravity function enables the 3D cursor to snap to vertices, curves and surfaces in the scene. An alignment object(line, plane or sphere) can be placed at an object's vertex or control point to provide the user with the ruler and compass in the 3D space. The 3D cursor can also snap to an alignment object. Interactive transformations follow the motion of the 3D cursor. As the 3D cursor continues to snap to scene objects and alignment objects during transformations, objects can be placed precisely.

All the above problems remain unsolved. But some suggestions have been given to improve the environment editors. The author leaves them for interested researchers.

Appendix A

The Database Schema

```
TYPE      transForm
TYPE      vector
TYPE      behaviortable
TYPE      space_list      LIST
TYPE      obj_inst_list   LIST
TYPE      master_list     LIST
TYPE      comp_list       LIST
TYPE      child_list      LIST

FRAME     environment
              name = " ";
              spaces = 0;
              space_list;
              obj_insts = 0;
              obj_inst_list;
              masters = 0;
              master_list;
              composites = 0;
              comp_list;
              next_tag = 1;
END
```

```
FRAME   space          META
            fheader = 1;
            name = " ";
            t_matrix;
            cur_rotate;
            cur_scale;
            cur_translate;
            boundary = -1;
            behavior = 0;
            contents = -1;
            in_environment = 0;
END
FRAME   master         META
            fheader = 2;
            name = " ";
            definition = -1;
END
FRAME   instance       META
            fheader = 3;
            master_frame;
            next = -1;
END
FRAME   composite      META
            fheader = 4;
            children = 0;
            child_list;
END
FRAME   child_frame    META
            fheader = 5;
            t_matrix;
            cur_rotate;
            cur_scale;
            cur_translate;
            definition = -1;
END
```

```
FRAME     primitive        META
           fheader = 6;
           file_name  = " ";
           prim_tag;
END
FRAME     obj_inst         META
           fheader = 7;
           name  = " ";
           prototype_name = " ";
           definition = -1;
END
```

Appendix B

User Manual

The environment editor is a tool that provides the artists with functions for creating and modifying virtual environments. The menu trees of the environment editor, shown in Figure 9, Figure 10 and Figure 11, reflect the system structure of the editor. Environment creation and environment modification are two major functions of the environment editor. The "Create Environment" submenu and the "Remodel Environment" submenu invoke these two logically independent parts of the environment editor(see Figure 9). The user interface of the environment editor has been described partially in the previous chapters. This appendix mainly focuses on explaining the sequences for generating a new environment or modifying an existing environment.

B.1 Environment Creation

To create an environment, the first step is to enter the name of the new environment from the text area of the screen. An environment is composed of a number of spaces. Each space contains a number of objects. So the sequence for constructing an environment should start with creating objects. Thus, the object editor is invoked.

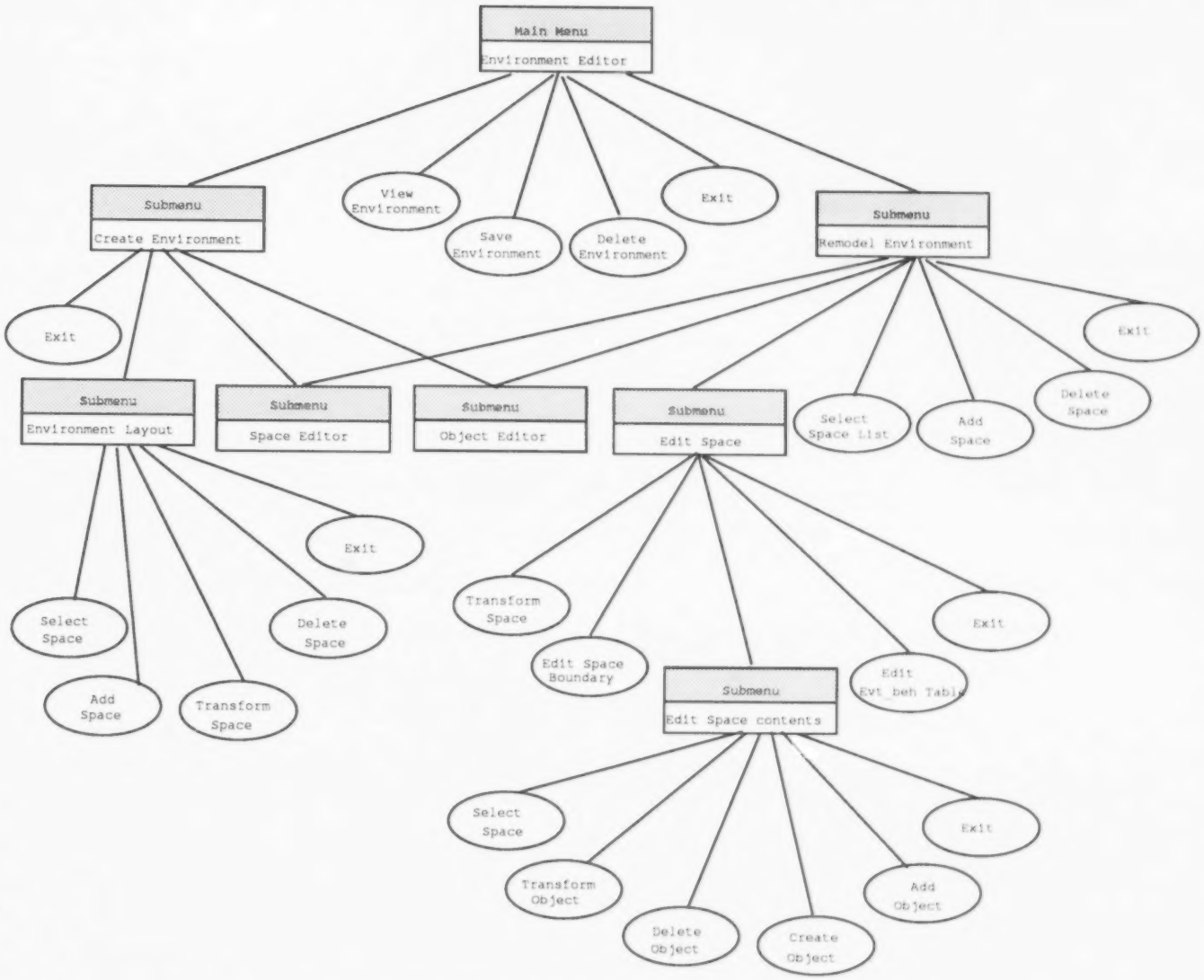

Figure 9: The high level menu structure of the environment editor.

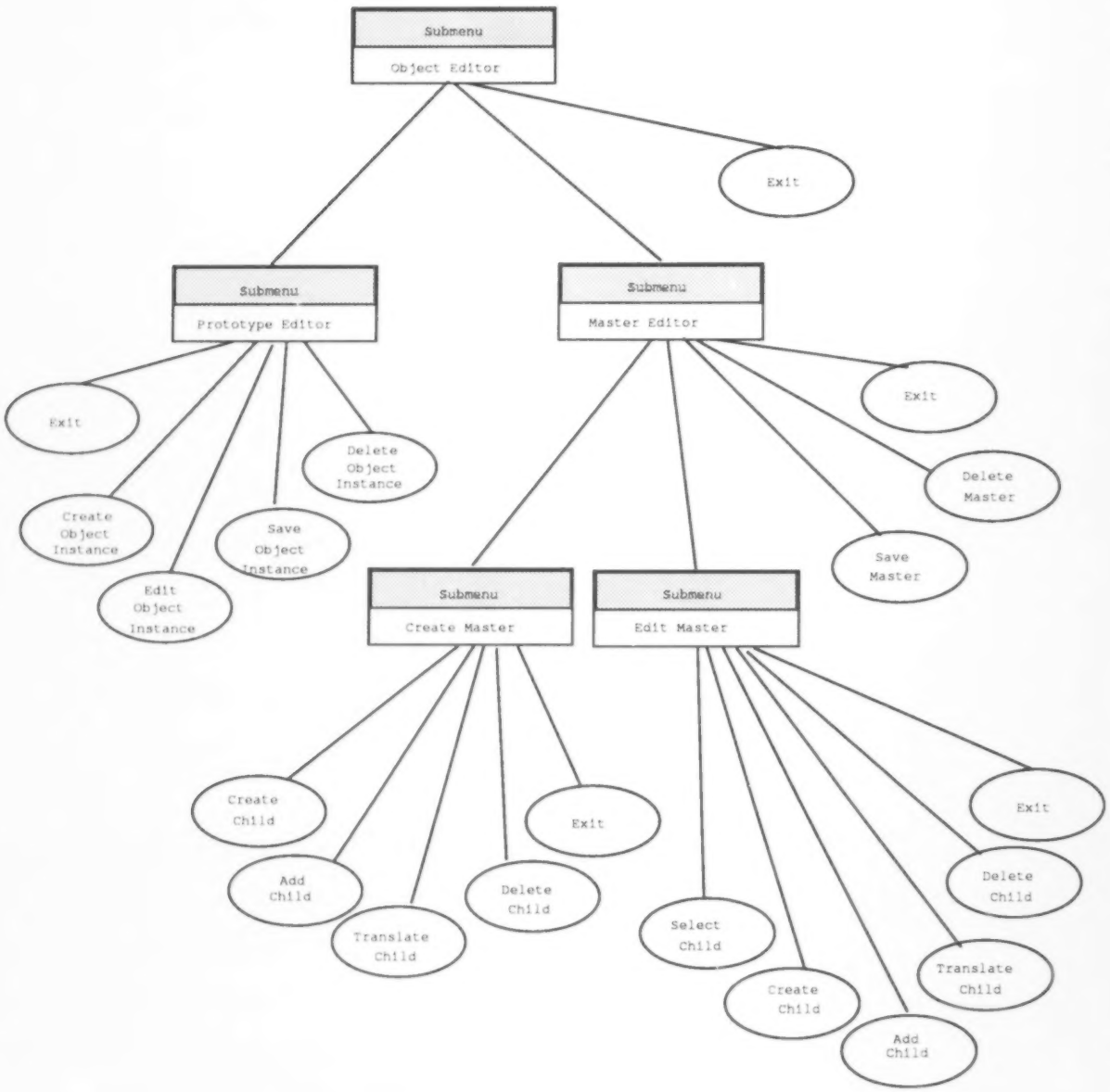

Figure 10: The menu structure of the object editor.

Figure 11: The menu structure of the space editor.

B.1.1 Use of Object Editor

The object editor is able to create a list of masters which will be used to create objects in an environment. The object editor has two parts - the object prototype editor and the master editor. The object prototype editor is responsible for creating and editing a list of object instances generated from the object prototypes. The master editor is responsible for creating and editing a list of masters created from the object instances and the existing masters.

The first step to object creation is invoking the object prototype editor and creating a list of object instances. The object prototype editor provides the means to assign and edit the parameters of an object prototype. After assigning the parameters of a selected object prototype, an object instance is automatically generated by the environment editor.

The master editor is used to create and edit masters. A master consists of a list of children, each of which can be created by the following steps. The user first selects the type of child to be created, either object instance or existing master, then selects an entity from the corresponding list. The left mouse button is used to select different entities. The selected entity is displayed in the editing area. Once a desired entity is found, press the middle mouse button to exit the selection mode. Then the child can be placed in the master by selecting the "Add Child" menu item. After a child is added to the master, it still can be transformed or deleted. The above procedures can be repeated a number of times to create a list of children for a master. Then the menu item "Save Master" must be used to append the new master to the master list of an environment. Editing an existing master involves editing the children of the master or adding new children to the master. So the first thing in master editing is to select a child from the master using the DataGlove. A selected child is flashed once when it is grabbed. A selected child can be transformed or deleted from the master. To add new children to the master, use the same procedure as master creation.

B.1.2 Use of Space Editor

After a list of masters have been created by the object editor, the user needs to exit the object editor and go back the environment creation submenu. Then the space editor can be invoked to create the boundary of a space, lay out the objects in the space and define the event_behavior table associated with the space. For simplicity, all the space boundaries are convex cylinders. To create a space boundary, two potentiometers are used to enter the minimum and maximum z values for the space, then a grid is used to enter the cross section parallel to the x-y plane. The user enters a convex polygon for the cross section. Then the name of the space is entered and the space is appended to the space list of the environment by selecting the "Save Space Boundary" menu item.

The space content layout submenu is used to fill a selected space with a number of objects. To do so, the user first selects a space boundary. Then the user selects a master from the master list, creates an instance of the master and positions it within the space boundary. The instance thus becomes an object in the space. The object is checked by the environment editor to see if it is internal to the space boundary. The environment editor notifies the user from the text area if the object is not internal to the space boundary. The user can adjust the object and place it in interior of the space boundary. After all the desired objects have been laid out in the space, the "Save Space Contents" menu item must be called to store the space contents in the environment database. After selection by the DataGlove or from the object list, an object becomes an active entity in the space and can be deleted or transformed.

The menu item "Define Event-Behavior Table" should be called after the space has been filled with objects. The user interface to the event_behavior table has been described in Section 3.3.3.2.

B.1.3 Environment Layout

The last step of the environment creation is to lay out the environment with the spaces that have been created. First a space is selected from the space list. Then it is placed in the environment. A space in the environment can be transformed or deleted. To save the created environment on the disk, press the "Save Environment" menu item.

B.2 Environment Modification

Environment modification can be the following operations - editing a space in the environment, deleting a space from the environment or adding a new space to the environment. Editing a space in the environment includes the following operations - transforming the entire space, editing the space boundary, editing space contents, and editing the event_behavior table associated with the space.

There are two ways of editing a space in an environment. One is editing the space in the environment coordinates by invoking "edit space" submenu. The other way is invoking the space editor since the "Remodel Environment" submenu can directly access the space editor(see Figure 9). Thus, a space can be selected from the space list and modified in the space coordinate system. The modification of the space will be automatically propagated to the environment. Also a new space can be created and appended to the space list of the environment and can then be placed in the environment.

Objects in an environment can be transformed or deleted, and new objects can be added to an environment by invoking the "Edit Space Contents" submenu(see Figure 9). In order to modify the internal structure of an object in the environment, the user has to modify the internal structure of the master from which the object is derived. Thus, the object editor must be invoked. Again, the "Remodel Environment"

submenu can directly access the object editor. The modifications of the master will be automatically propagated to all the objects that it derives.

Bibliography

[BB71] J.J. Batter and F.P. Jr. Brooks. Grope-i: A computer display to the sense of feel. In *Information Processing, Proc. IFIP Congress*, pages 759–763, 1971.

[Bie90] E. A. Bier. Snap-dragging in three dimensions. *Computer Graphics*, 24(2):193–204, 1990.

[Bol80] R. A. Bolt. Put-that-there: Voice and gesture at the graphics interface. In *Proceedings of Siggraph*, pages 262–270, 1980.

[Bol81] R. A. Bolt. Gaze-orchestrated dynamic windows. In *Proceedings of Siggraph*, pages 109–119, 1981.

[Bro77] F. P. Jr. Brooks. The computer scientist as toolsmith: Studies in interactive computer graphics. *Information Processing*, pages 625–634, 1977.

[Bro86] F. P. Jr. Brooks. Walkthrough - a dynamic graphics system for simulating virtual buildings. In *Workshop on Interactive 3D Graphics*, pages 9–21, 1986.

[CB+90] Y. Harvill C. Blanchard, S. Burgess et al. Reality built for two: A virtual reality tool. In *Proceedings on Interactive 3D Graphics*, pages 35–36, 1990.

[Dep89] Dept. of Computing Science, Univ. of Alberta. *GMP: A General Purpose Modeling Package*, 1989.

[Dep90a] Dept. of Computing Science, Univ. of Alberta. *FDB: A Frame Based Database System*, 1990.

[Dep90b] Dept. of Computing Science, Univ. of Alberta. *Minimal Reality(MR): A Toolkit for Virtual Reality Applications*, 1990.

[FPJB+90] J. J. Batter F. P. Jr. Brooks, M. Ouh-Young et al. Project grope - haptic displays for scientific visualization. *Computer Graphics*, 24(4):177–185, 1990.

[Gre90] M. Green. An artistic scenario for virtual reality. Technical report, Univ. of Alberta, Edmonton, AB T6G 2H1, 1990.

[Gre91] M. Green. Environment editor specification. Technical report, Univ. of Alberta, Edmonton, AB T6G 2H1, 1991.

[Gre92] M. Green. Object modeling language specification. Technical report, Univ. of Alberta, Edmonton, AB T6G 2H1, 1992.

[Gre 7] M. Green. The virtual reality papers, volume 1. Technical report, Univ. of Alberta, Edmonton, AB T6G 2H1, CS-TR-90-7.

[Iwa90] H. Iwata. Artificial reality with force-feedback: development of desk-top virtual space with compact master manipulator. In *Proceedings of Siggraph*, 1990.

[JPGT89] N. M. Thalmann J. P. Gourret and D. Thalmann. Simulation of object and human skin deformations in a grasping task. In *Proceedings of Siggraph*, 1989.

[Kru83] M. W. Krueger. *Artificial Reality.* Addison-Wesley Publishing Company, 1983.

[Kru91] M. W. Krueger. *Artificial Reality II.* Addison-Wesley Publishing Company, 1991.

[MWKH85] T. Gionfriddo M. W. Krueger and K. Hinrichsen. Videoplace - an artificial reality. In *CHI Proceedings*, pages 35–40, 1985.

[RG91] H. Rijpkema and M. Girard. Computer animation of knowledge-based human grasping. *Computer Graphics*, 25(4):339–348, 1991.

[Sun92] H. Sun. *A Relation Model for Animating Natural Behavior in Dynamic Environments.* PhD thesis, Univ. of Alberta, 1992.